Document made available under the Patent Cooperation Treaty (PCT)

International application number: PCT/US05/007278

International filing date: 04 March 2005 (04.03.2005)

Document type: Certified copy of priority document

Document details: Country/Office: US
Number: 60/549,946
Filing date: 04 March 2004 (04.03.2004)

Date of receipt at the International Bureau: 18 April 2005 (18.04.2005)

Remark: Priority document submitted or transmitted to the International Bureau in compliance with Rule 17.1(a) or (b)

World Intellectual Property Organization (WIPO) - Geneva, Switzerland
Organisation Mondiale de la Propriété Intellectuelle (OMPI) - Genève, Suisse

1305366

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

April 07, 2005

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE.

APPLICATION NUMBER: *60/549,946*
FILING DATE: *March 04, 2004*
RELATED PCT APPLICATION NUMBER: *PCT/US05/07278*

Certified by

Jon W. Dudas

Under Secretary of Commerce
for Intellectual Property
and Director of the United States
Patent and Trademark Office

Rev 02/04

16179 U.S. PTO 030404

22151 U.S. PTO 60/549946 030404

PROVISIONAL APPLICATION COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION under 37 CFR 1.53 (c).

Express Mail No.: EV044751241US

MAIL STOP PROVISIONAL APPLICATION
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Docket Number: BU-115Xq800	**Type a Plus sign (+) inside this box →**	+

INVENTOR(s)/APPLICANT(s)

LAST NAME	FIRST NAME	MIDDLE INITIAL	RESIDENCE (CITY AND EITHER STATE OR FOREIGN COUNTRY)
Herbordt	Martin	C.	29 Aqueduct Road, Wayland, MA 01778
Van Court	Thomas		3 Marshall Place, Charlestown, MA 02129

[ ] Additional Inventors are being named on Page 2 attached.

TITLE OF THE INVENTION (280 characters max)

SYSTEM AND METHOD FOR PROGRAMMABLE-LOGIC ACCELERATION OF BIOIONFORMATICS AND COMPUTATIONAL BIOLOGY APPLICATIONS

CORRESPONDENCE ADDRESS

[X] **Customer Number 207**

which is associated with the Law Firm of:
WEINGARTEN, SCHURGIN, GAGNEBIN & LEBOVICI LLP
Ten Post Office Square
Boston, Massachusetts 02109
United States
Telephone: (617) 542-2290 Fax: (617) 451-0313

ENCLOSED APPLICATION PARTS *(CHECK ALL THAT APPLY)*

[X] Specification incl. Figs. Number of pages [152]

[ ] Drawing(s) Number of sheets [ ]

[X] Small Entity status is entitled to be, and hereby is, asserted for this application

[ ] Other (specify)

METHOD OF PAYMENT *(CHECK ONE)*

[X] A check in the amount of $80.00 is enclosed to cover the Provisional Filing Fee

[ ] The Commissioner is hereby authorized to charge filing fees and credit Deposit Account Number 23-0804

Please recognize the following attorneys with powers in this application.

Stanley M. Schurgin, Reg. No. 20,979
Charles L. Gagnebin III, Reg. No. 25,467
Victor B. Lebovici, Reg. No. 30,864
Beverly E. Hjorth, Reg. No. 32,033
Holliday C. Heine, Reg. No. 34,346
Gordon R. Moriarty, Reg. No. 38,973
James F. Thompson, Reg. No. 36,699
Richard E. Gamache, Reg. No. 39,196

Respectfully submitted,

SIGNATURE James F. Thompson DATE March 4, 2004

TYPED or PRINTED NAME: James F. Thompson REGISTRATION NO. 36,699

PROVISIONAL APPLICATION FILING ONLY

JFT/raw 302854-1

UNITED STATES PROVISIONAL PATENT APPLICATION

of

Martin C. Herbordt and Thomas Van Court

for

System and Method for Programmable-Logic acceleration of Bioinformatics and Computational Biology applications

Express Mail Number
EV044751241US

PROGRAMMABLE DEVICES FOR BCB APPLICATIONS

STATEMENTS REGARDING FEDERALLY SPONSORED RESEARCH

Not applicable.

CROSS-REFERENCE TO RELATED APPLICATIONS

Not applicable.

FIELD OF THE INVENTION

This invention relates generally to programmable electronic devices and more particularly to a method and apparatus for utilizing programmable devices in bioinformatics and computational biology (BCB) applications.

BACKGROUND OF THE INVENTION

As is known in the art, biologists have devised methods for accumulating large amounts of genetic DNA sequence information. Because considerable information may be found by comparing the DNA sequences between different genes and organisms, major efforts are underway to sequence the DNA from many different human individuals, and many different species. Due to the high speed of modern sequencing technology, biologists accumulate data at a rate which greatly exceeds known techniques for analyzing the data. The biological sequences, such as DNA, RNA or protein sequences are typically stored as linear arrays of characters where each character corresponds to a base element or residue of the particular sequence. Such linear character arrays are typically referred to as strings.

As is also known in the art, the term "bioinformatics" refers to the use of computer processors and computer related technology to automate the process of interpreting the above-mentioned large amounts of data. To efficiently process the large amounts of biological data, algorithms referred to as bioinformatics algorithms have been developed.

Bioinformatics algorithms have been developed to compare full or partial sequences depending upon the particular biological problem being addressed. Often a choice of several algorithms exists for solving a given problem. The algorithms can be implemented using general purpose processors, specially designed custom integrated circuits (ICs) or programmable devices.

General-purpose computer processors are designed for flexibility and thus are useful for a wide variety of applications. One problem with using general-purpose computer processors, however, is the relatively slow speed at which they can process the large amounts of data.

Specially designed custom ICs can be optimized to rapidly implement functions required in a specific bio-informatics algorithm or a specific biological problem. One problem with this approach, however is that custom ICs are relatively expensive and require a relatively long lead time to fabricate. Furthermore, it is often desirable to try more than one algorithm to solve the same problem or it may be desirable to try multiple variants of one algorithm on the same problem. Thus, the custom IC approach can be quite costly and time consuming and does not provide flexibility which is often desired.

Programmable devices such as field programmable gate array (FPGA) circuits can be used to facilitate high-speed processing of fixed algorithms. FPGA circuits are composed of a plurality of simple logical elements, and programmable means to dynamically rewire the connections between the various logical elements to perform different specialized logical tasks. Often this is accomplished by fuse-antifuse circuit elements (or gates) that connect the various FPGA logical circuits. These fuse-antifuse elements can be reconfigured by applying appropriate electrical energy to the FPGA external connectors, causing the internal FPGA logical elements to be connected in the appropriate manner.

An alternate way to produce custom integrated circuit chips suitable for the

implementation of custom algorithms is by more standard chip production techniques, in which a fixed logical circuit is designed into the very production masks used to produce the chip. Because such chips are designed from the beginning to implement a particular algorithm, they often can be run at a higher logic density, or faster speed, than more general purpose FPGA chips, which by design contain many logical gates that will later prove to be redundant to any particular application.

Several exemplary prior art are systems are commercially available. For example, one system based upon based upon custom VLSI sequence analysis chips is the GeneMatcher2™ genetic analysis system manufactured by Paracel Corporation (1055 East Colorado Boulevard, Fifth Floor Pasadena, CA 91106-2341). Another system, based upon custom FPGA chip technology, is the DeCypher™ genetic analysis system manufactured by TimeLogic Corporation (1914 Palomar Oaks Way, Suite 150, Carlsbad CA 92008). Still another system based upon FPGA chip technology is the SysGen system manufactured by XiLinx (2100 Logic Drive , San Jose, CA 95124-3400).

It would, however, be desirable to provide a system and technique which can rapidly implement bioinformatics algorithms and which allows a user to implement a number of different algorithms in a relatively short period of time. It would be further desirable to provide a relatively inexpensive processing platform (e.g. a personal computer) which can be used to rapidly perform computations for BCB applications. It would also be desirable to provide a system which provides the function which FPGA-based turn-key products provide, but at lower cost and with the capability to extend the utility to many, if not most, BCB computations.

SUMMARY OF THE INVENTION

In accordance with the present invention, a system that enables bioinformatics and computational biology (BCB) researchers to create, with a modicum of effort, field programmable gate array (FPGA) configurations (circuit designs) that yield speed-ups of a factor of 1000 and greater for a wide variety of BCB applications is described. The system receives inputs for a particular application from a user and performs two transformations: (1) problem specification to circuit specification; and (2) circuit specification to circuit implementation. The users can be bioinformatics practitioners or computational support staff in bioinformatics groups. It should be appreciated that the circuit specification to circuit implementation transformation is normally the realm of a circuit designer. Thus, the system of the present invention allows a user without circuit design expertise (e.g. a bioinformatics practitioner or computational support person in a bioinformatics group) to generate circuits for use in a bioinformatics application.

The problem specification to circuit specification transformation involves creating algorithms. It should be appreciated that problem specification to circuit specification transformation is much more than simply mapping an existing algorithm to hardware. Invariably, this transformation means changing the algorithm. This is because existing programs or algorithm specifications cannot simply be transformed to a circuit specification since the target hardware, i.e. an FPGA, is fundamentally different from a standard computer (e.g. a PC, workstation, or cluster). And, for many applications, fundamentally different hardware requires a fundamentally different algorithm to obtain maximal (or even tolerable) performance. Thus, by transforming the application into an appropriate FPGA algorithm to create a circuit specification and transforming the circuit specification into an efficient circuit allows non circuit designers to generate effective circuits for use in BCB applications.

Importantly, the system of the present invention automates the efficient allocation of resources in ways specific to the use of FPGAs for computation. This is in addition to optimizations already done during synthesis and place-and-route functions. One example of such automation is referred to as "precision management" which corresponds to a process for

correctly-sizing computational units to minimize their chip area and maximize the potential for parallelism. Another example of such automation is speed-matching (also referred to as rate balancing) among phases of the computation. This is accomplished by replicating units appropriately to avoid bottlenecks in data flow.

The system of the present invention thus provides functions and modules that bridge the gap between the programmer and the FPGA. To the programmer, the functions and modules look like high level language (HLL) function calls or GUI computational blocks. At a lower level, however, functions and modules contain circuit specifications optimized to FPGAs. These functions and modules require creation of FPGA-specific algorithms in addition to circuit designs.

The system also includes a set of templates to construct application specific modules as well as constraints and features that allow the use of standard compiler techniques in creating high-quality circuits by non-circuit-designers. Such templates, constraints and features are included in the design of a computer language. The system also includes a compiler which includes software components (e.g. a flow analysis/precision management processor and a parallelism/resource balancing processor) that enable the creation of high-quality circuits by non-circuit-designers.

Thus, in accordance with the present invention, a system for providing an FPGA circuit includes a low-cost FPGA-based computational coprocessor includes a compiler, a set of domain specific policies and rules for user defined custom applications, a set of hardware contexts and application independent but hardware dependent code. The compiler performs flow analysis/ precision management operations, maps hardware resources and determines the amount of parallelism which can be included in a circuit. The compiler also performs a resource balancing operation and provides code which can be used to program an FPGA circuit.

With this particular arrangement, a relatively inexpensive processing system for rapidly processing data in bioinformatics applications is provided. Thus, a system provided in accordance with the present invention brings to an individual researcher a relatively low cost, rapid, convenient and flexibility computation capability. This approach allows a conventional PC to be augmented with an FPGA-based computational coprocessor provided as a plug-in board similar in cost and ease of installation to a graphics card. The coprocessor is provided with a computational environment that enables its efficient use. This environment supports a wide range of user sophistication by including: (1) standard applications (such as BLAST) for users with no computer background; and (2) Application/Function libraries for users with scripting or spread-sheet programming capability; (3) a high-level language for programmers, and (4) a Hardware Description Language (HDL) interface for FPGA/circuit designers. In a preferred embodiment, the system can provide a factor of 100 to 1000 speed-up over a PC with a materials and manufacturing cost of between $2000 and $5000.

The system of the present invention will not only allow current large-scale applications to be run on a relatively low cost processing platform (e.g. a desk-top processor), it will break the chicken-and-egg phenomenon in algorithm development whereby most application developers in the BCB field avoid computationally complex algorithms because the appropriate hardware is either not easily available or not available at all. The present invention makes available a system which allows the community of developers to apply algorithms with complexity greater than $O(N^2)$ which is the nominal limit for everyday PC computations with the expected data sets.

A further advantage of flexibility is derived from the fact that bioinformatics and computational biology do not occur in a vacuum. That is, it is not generally possible solve a given problem via brute force computation. Rather, the norm is develop and run computations in close collaboration with biologists. A typical scenario involves a feedback loop where biologists formulate problems and interpret results while computer specialists create and modify the applications as directed by the underlying biology. Essential to the

functioning of this development loop is agile computing which is provided by the present invention.

Although this invention finds application in desktop PC computing, a system provided in accordance with the techniques described herein is equally applicable to clusters of PCs. In that case, the goal is to use the coprocessor to address complex and intensive computations such as those required in molecular dynamics. The potential impact therefore is (at least) two-fold: (1) to empower the individual researcher by reducing "heroic"

computations that would normally take weeks to minutes and (2) to create knowledge through brute computational capability.

Empowering the individual researcher by reducing "heroic" computations that would normally take weeks to minutes has two consequences. First, cost reduction (supercomputer on the desktop) and second, the potential for developing new and qualitatively different algorithms.

The aspects of BCB addressed by the present invention include analytic applications. Some of the broad domains within analytic applications are: genomics, functional genomics, proteomics, pharmacogenomics, biosimulation, combinatorial chemistry, and drug design. Within each of these domains are many applications; e.g. a sample from genomics includes genome assembly, gene identification searches, motif searches, genome characterization, and comparative genomics. Applications often share computational characteristics with others within and across domains, but also often vary greatly even within a domain.

With the completion of the human genome project, it is generally agreed that emphasis will move from description to function, structure, dynamics, and systems. These new problems and new technologies generally require different classes of algorithms than those central to genomics which is dominated by string processing, especially classical string algorithms, suffix tree algorithms, and dynamic programming. Classes of algorithm that

appear to be central to the new emphasis are searching high-dimensional parameter spaces, linear algebra, and molecular dynamics.

The system of the present invention includes a hardware/software compute engine installable and upgradeable in a PC by a lay person. Ideally, the hardware is completely transparent with the software providing usability at the level appropriate to the user capability. Although more user effort and experience will yield broader product utilization, the system of the present invention should find broad applicability for all those persons performing BCB computations.

As mentioned above, the present invention includes both hardware and software components. In one embodiment, the hardware corresponds to a high-performance FPGA chip disposed on a circuit board having a standard high-speed (PCI) interface. An existing third party circuit board (e.g. one of the types provided from Avnet or Nallatech) may be sufficient. However, in some applications, a special memory or I/O interface may be necessary or desired, in which case a custom board could be designed. The software interface can be provided as a PCI driver and standard GUI.

The present invention also contemplates a system which includes a hierarchy of solutions. The particular solution to select from the hierarchy of solutions depends upon a variety of factors including user sophistication. Such a hierarchy of solutions thus includes but is not limited to the following set of solutions:

(a) a solution in which complete applications for users with almost no computer background are provided (e.g. the level of support provided by TimeLogic as well as the same applications could be provided in this level of solution);

(b) a solution in which application libraries and a design flow editor for users who need more BCB applications than the ten or so now provided in conventional systems (this component is intended to provide a product for BCB analogous to Xilinx's System Generator for DSP). The function modules to be provided include but are not limited to function modules for data access, string processing, microarray analysis, linear

algebra, and associative operations. At this level, users should be able to mix-and-match components to create a large number of custom applications;

(c) a solution in which users are allowed to create their own modules (e.g. via a domain specific high-level language that compiles to the hardware (the idea is to create for BCB what SA-C provides for computer vision hardware);

(d) a solution which includes a hardware description language interface for users with moderate hardware sophistication, e.g. at least an undergraduate degree in computer/electrical engineering with an interest in creating highly tuned components, or modules for distribution or sale.

The present invention can be used in a variety of different applications including but not limited to Genomics, Proteomics, Partial string matching, Bayesian Inference, Search, Linear Algebra and statistical techniques. The system of present invention may be found to be particularly useful in solving problems having characteristics including but not limited to: (1) data sets that fit on a chip (at least for local computation) e.g. 1-100 million bits at a time; and (2) relatively simple, highly repetitive evaluation functions.

One example of a suitable problem is that of using microarrays to perform thousands of experiments and then correlating activity as measured by the microarrays to provide a binary outcome. For example, it may be desirable to determine which very small subset of genes (e.g. 3 genes) correlates the best with the outcomes. That is, for all combinations of 3 genes out of some thousands of genes with some thousands of experiments, it is desired to find the combination that lends itself best to being a discriminator. In this case, fast computation is achieved by using dot products which are done by streaming vectors through multiply accumulate units. The present invention utilizes easily constructed or obtained hardware (PCI card, FPGA, memory, PC) which is programmed to allow the hardware to rapidly perform such computations.

Parallelism and reuse

Reuse has three possibilities: (1) use many times in succession so compute once and keep on chip; (2) use many times, but distributed and if much work to compute, then compute once and keep off chip; and (3) use many times, but distributed and if easy to compute, then compute each time.

Parallelism (i.e. computing in parallel) can be done many ways as well: (1) depending upon reuse more or less computing may be called for. Would create a gigantic dot-product computer where we stream on the 1000 by 1000 vectors and compute all 1000000 dot products simultaneously. Actually a subset depending on # of I/O pins, computing area, pin bandwidth, etc.; and (2) to keep results from dot-product on chip as long as possible, may be good to do inversion before off-loading. Can partially reconfigure, or pipeline dot-product computation with inversion.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing features of this invention, as well as the invention itself, may be more fully understood from the following description of the drawings in which:

FIG. 1 is a block diagram of a system for providing code to program an FPGA circuit;

FIG. 1A is a block diagram of a compiler which performs at least one of flow analysis/ precision management operations, mapping of hardware resources, determining the amount of parallelism which can be included in a circuit and resource balancing;

FIG. 2 is a block diagram of a system which performs BCB computations using a hardware accelerator provided from an FPGA coprocessor circuit;

FIG. 3 is a block diagram of a template included in a system which provides code to program an FPGA circuit in BCB computations;

FIG. 4 is a block diagram of a series of software components which are coupled to perform a function in a BCB application;

FIG. 5 is a block diagram of a series of software components which provide an illustrative example of parallelism and rate balancing;

FIG. 6 is a block diagram of a series of processing units which provide an illustrate

example of parallelism and rate balancing;

FIG. 7 is a block diagram of an exemplary system which utilizes distribution and collection networks in computations for a BCB application; and

FIGs. 8 and 8A are a series of block diagrams which illustrate a circuit which implements an optimization technique for a hill-climbing function.

DETAILED DESCRIPTION OF THE INVENTION

The present invention relates to the use of field programmable gate arrays (FPGAs) in BCB applications. FPGAs are integrated circuits whose (apparent) circuitry can be determined, or programmed, in the field. This is in contrast to application specific integrated circuits (ASICs), for example, which have circuitry which is fixed at fabrication time. The trade-off for this flexibility is that FPGAs are less dense and thus typically slower than ASICs. For many applications, however, the reduced engineering and manufacturing cost and drastically improved time-to-market offered by FPGAs more than make up for the drawbacks. Beyond the configurable substrate, current generation, high-end FPGAs can also contain optimized modules, including entire microprocessors.

FPGAs, however, present a computational model that is fundamentally different than that presented by a PC or other standard computer. What this means is that, for any application, it is likely that the algorithm for its optimal execution will be different for an FPGA than it is for a PC. In particular, an FPGA algorithm requires software constructs or functions while the PC algorithm will not. Also, applications must use these constructs to obtain maximal performance. Consequences of this are discussed below.

In one aspect of the present invention, the system of the present invention relates to a method and apparatus for providing an FPGA circuit use in a bioinformatics and computational biology (BCB) application. The system include a series of templates or software components which can be customized by a user to implement a particular function in an FPGA circuit which is adapted for use in a BCB application. The system automatically analyzes a variety of factors to determine a number of functional blocks which can rapidly be

used on the FPGA and the system also automatically determines the degree of parallelism and the width of certain data which can be implemented in the FPGA such that the FPGA can rapidly perform BCB computations.

In another aspect of the present invention, a system which includes an FPGA co-processor circuit adapted for processing BCB applications, is described.

Referring now to FIG. 1, a system 10 for generating FPGA circuit code includes a user programming interface 14 which accepts user input 12. One exemplary user interface is described in detail herein below. The interface 14 provides the user input to a processing system 16. Processing system 16 also includes a compiler 18 the details of which will be discussed below in conjunction with FIG. 1A. Suffice it hear to say, however, that the compiler 18 performs one or more of the following operations: flow analysis/ precision management operations; mapping of hardware resources and determining the amount of parallelism which can be included in a circuit, resource balancing and provides code for an FPGA circuit.

The compiler receives domain specific polices 20a

, content which is applicable to all applications in all hardware contexts 20b (i.e. it will apply regardless of the FPGA chip family being used or the type of board carrying the chip)

and material 20c which is independent of any application (and thus can be used for any application) but which is dependent upon the particular type of hardware used in the system.

That is, for any combination of FPGA chip and circuit board which will be used, there will be corresponding information in storage 20c which matches the specific hardware embodiment. Thus, the processing system includes application specific content, application independent-hardware independent content and application independent-hardware specific content. It should be appreciated that some of the content in each of boxes 20a-20c may be represented

as data and some as executable code and that the particular manner in which the content represented (e.g. as data or code) is selected for ease of use in the system.

The compiler 18 provides two outputs: (1) host interface code and (2) high level language (HLL) code which can be fed to a conventional compiler 22 which provides appropriate FPGA circuit code 23 which can be loaded onto an FPGA (e.g. FPGA circuit 26b in FIG. 2) using any one of the many techniques known to those of ordinary skill in the art. It should be appreciated that compiler 22 includes conventional synthesis tools and others tools as is generally known.

Referring now to FIG. 1A, the compiler 18 includes an interface 18a (which may be provided for example as a GUI / text interface) which receives user input values (e.g. from user 12 in FIG. 1) and provides the values to a parser/converter 18b. It should be understood that the user can be a bioinformatics practitioners or a computational support staff person in a bioinformatics groups. It should be also appreciated that a circuit specification to circuit implementation transformation is normally the realm of a circuit designer. However, in the system described herein, once the user inputs certain information, the system will automatically generate a circuit specification. Thus, the system of the present invention allows a user without circuit design expertise (e.g. a bioinformatics practitioner or a computational support person in a bioinformatics group) to generate circuits for use in a bioinformatics application by supplying limited a amount of information.

The parser/converter receives the text and general block diagrams which have been entered and converts this information to an internal representation so that different mathematical operations can be identified (e.g. multiplication operations, addition operations, etc...) and so that the process of connecting different blocks to each other can begin and so that the process of building a model based upon what the user has presented can begin. The parser/converter thus receives user input and parses the textual and graphical input and converts the input to an internal representation (i.e. an internal representation of adder

circuits, multiplier circuits, etc...) which is then provided to a flow analysis precision management processor 18c. It should be appreciated that parser/converter performs technology independent transformations.

The flow analysis precision management processor 18c receives the information fed thereto and analyzes the information fed thereto and determines certain circuit characteristics (e.g. bit width allocation, bit precision, etc...) based upon the analysis. That is, flow analysis precision management processor analyzes the basic properties of the arithmetic which will be performed. Thus, flow analysis precision management processor performs a precision management function as well as other functions and at the flow analysis precision management processor output, the number of bits which will be used at every part in the calculation is known. It should also be appreciated that the flow analysis precision management processor performs technology independent transformations.

The flow analysis precision management processor provides the information generated therein to a hardware resource mapper 18d. The hardware resource mapper examines the different arithmetic operations and identifies appropriate hardware to carry out the operation. For example, if the hardware resource mapper identified a small multiply (e.g. a 2 bit times 2 bit multiply) and a large multiply (e.g. a 12 bit times 14 bit multiply) then the mapper may decide that the large multiple can use special hardware if it is available and the small multiply will not benefit from the special hardware. Thus, the mapper begins taking the hardware resources (e.g. knowledge that there is a special multiplication hardware) and mapping the abstract logic into the specific hardware resources. Thus, at this point the compiler system has generated a "thumbnail sketch" of all of the hardware resources which will be required.

It should be appreciated that the mapper has some technology dependencies (e.g. is a hardware multiplier available) and that the output of the hardware mapper it is known (to at least a first approximation) which hardware resources that each user-specified unit will require as well as a thumbnail estimate for timing requirements for each of those units.

The hardware resource mapper 18d provides the information generated therein to a parallelism / resource balancing processor 18c. Processor 18c also has technology dependencies as inputs. As will be described in further detail below in conjunction with FIGs. 5-7, parallelism / resource balancing processor 18c determines, among other things, the number of parallel processing units of a particular type which will be used in a circuit as well as the types of distribution and collection circuits which will be used to achieve rate balancing. It should be understood that time is considered a resource within the context of this processing. Thus, after this processing step, parallelism has been allocated and rate balancing is complete.

Management of Parallel Vector Access

In microarray analysis, for example, many important techniques are based upon examining combinations of thousands of vectors. When the size of the combination is >2, the number of combinations is very large, although the number of actual vectors is quite manageable. In programmable circuits, it is possible to perform hundreds of the computations in parallel making these computations tractable. The problem is to orchestrate the routing of the vectors from the storage to the parallel computation units so that combinations are available for processing at least once, but rarely more than that.

Datapath Width Sizing/Precision Management

Programmable circuits allow minimization of chip resources as a function of the computation. This allows chip resources to be used for other purposes, e.g. additional parallelism, additional computations, error checking, storage, etc. One minimization is in the number of bits in the datapath including ALUs, multipliers etc. from the 32 or 64 in a microprocessor down to something smaller, including sometimes a single bit. It is desirable to maximize the reduction while minimizing the risk that the computation will be compromised.

During the process, the arithmetic is analyzed using probability density functions to represent ranges of possible values or to represent uncertainty due to quantization errors. This is a superset of worst-case analysis; it gives much more statistical information and so allows for substantially more efficiency.

The parallelism / resource balancing processor 18c provides the information generated therein to a hardware description language (HDL) generator. The HDL generator receives the abstract concepts defined by the user-supplied application logic, the specific numbers of units that the hardware should be able to manage, the specific number of units involved in rate balancing, etc... and generates code which is suitable for use by the compiler 22 described above in conjunction with FIG. 1. The HDL output may be provided, for example, as Verilog HDL.

It should be appreciated that FIGs. 1 and 1A, illustrate the process for producing code needed to write a program to run on the FPGA coprocessor (i.e. FIGs. 1 and 1A illustrate the flow from user input to FPGA application code).

As will become apparent from the description of FIG 2 below, it should also be appreciated that there exists a parallel process for producing a program to run on a main or host processor in the PC. These two pipelines of software generation can proceed in parallel. One process runs from user input to the FPGA and the other process runs from user input to the host processor.

There exists an integration between both a hardware data path between the hardware FPGA and hardware host processor and also a software data path between the host application executing on the host processor and the FPGA application executing in the FPGA on the coprocessor.

Thus, as shown in FIG. 2, it should be appreciated that the computer program as a whole is executed on two separate processors, namely the host processor and the PFGA

coprocessor. That is, there exists one program executing on two different processors (i.e. the FPGA coprocessor and the host processor), with different parts of the program with different responsibilities running on each processor.

In FIG. 2 a host interface program is provided to a host processor 25 where it is executed as a host application.

In FIG. 1A, once the compiler operation is complete, the complier provides host interface code (e.g. to block 21 in FIG. 1) and compiler code to the compiler 22 (FIG. 1). The compiler 22 then produces the FPGA circuit code 23. The host interface code 21 and the FPGA circuit code 23 must be integrated. This is accomplished by the user writing a program. However, rather than using one programming language, the user will use two programming languages. First, the user can use a conventional (e.g. one of the C programming languages to build the host application program). Second, the user uses the programming language compilation tools described herein to build the FPGA part of the application. Thus the compiler 18 generates HDL code for compiler 22 to provide FPGA code as well as additional output (e.g. C language output), that will go to the host application development string. This creates the coupling between the software parts of the application (i.e. between the host part of the application and the FPGA part of the application).

Thus, the host interface code is used by the host application to communicate with the specific FPGA application. It should be understood that this code will be different for every FPGA application.

As is evident from FIG. 2, the FPGA circuit code 23 generated in FIG. 1A, is the content that gets loaded into the FPGA circuit 26b.

Turning now to FIG. 2, a system 24 for rapidly performing computations in a BCB application includes a processing platform 16' having a host processor 25 and an acceleration

circuit 26 which is here provided as an FPGA co-processor circuit board 26 coupled to the host processor and having disposed thereon an FPGA circuit 26b. The FPGA co-processor also includes a storage region 26a having stored therein characteristics of the FPGA board. Storage region 26a contains information which describes the individual characteristics of each different possible coprocessor board, even if that particular FPGA chip is constant across different coprocessor boards. Different coprocessor boards will have different communication resources to talk to the host, will have different onboard memory and similar kinds of resources.

The FPGA circuit code 23 generated by the system of FIG. 1 is loaded into the FPGA using any technique known to those of ordinary skill in the art to configure the FPGA to perform certain processing functions. The host interface code 21 (FIG. 1) generated by the system of FIG. 1 is supplemented with additional code to provide a host interface program 27 which is loaded into and executed by the host processor 25.

The special purpose FPGA co-processor corresponds to an accelerator which plugs into the processing platform. The accelerator is transparent to the user 12'. An analogy is a graphics card: although it provides an essential function, it is transparent to PC users with only graphics programmers (e.g. video game builders) accessing it directly. The hardware is supported through a language/function-library infrastructure to make the plug-in card as usable to the application developer as a standard high-level programming language. Usability requires that the end users be given the tools so that they can, with minimal training, use them.

In one embodiment, the hardware is a high-end FPGA chip 26b on a circuit board 26. Although not explicitly shown in FIG. 2, circuit board 26 includes a standard high-speed (PCI) interface. The circuit board, may for example, be provided as a commercially marketed circuit board (e.g. of the type available from Avnet or Nallatech) which can accept an FPGA

circuit. In some applications, however, it may be necessary to develop a circuit board having a special memory or I/O interface in which case a circuit board would to be designed. The software interface can be provided as a PCI driver and standard GUI.

The FPGA circuit 26b may thus contain new FPGA-specific algorithms (i.e., that have not already been created for other systems).

High-end FPGAs have the following characteristics:

1) Programmable in milliseconds by uploading the desired configuration. This also means that the FPGA can be reprogrammed for other applications just as quickly;

2) 4 million+ configurable gate-equivalents;

3) Millions of communication paths, both local and global;

4) Circuits are generally designed using hardware description languages (VHDL, Verilog); much available Electronic Design Automation (EDA) support

5) Design modules often available as "intellectual property" (IP) blocks

6) Hardwired on-chip gigabit interfaces (Infinib and, Gigabit Ethernet, etc.)

7) Hardwired on-chip microprocessors, busses, and memory modules

8) Hundreds of hardwired 18-bit multipliers

In architecture good performance is dictated by fast cycle times, but at least as important are parallelism and locality. In other words, how many processing elements (PEs) which can be simultaneously brought to bear on a certain computation and how long does it take to get the data to the PEs. It is often the case that these goals are at odds with one another. It has been found, however, that in the mapping of BCB to FPGAs they can often be achieved together. Some of the reasons are as follows.

Data elements have a small number of bits

This is true in sequences, microarrays, and even molecular dynamics when cast appropriately. Since only a few bits need to be processed per datum, FPGA elements can be configured into many thousands of Arithmetic Logic Units (ALUs).

This is in contrast to personal computers (PCs) where there are a small, fixed, number of large ALUs. FPGAs can also have optimized dedicated circuitry to support hundreds of parallel multiplications.

Data Set Sizes and Management

For many applications, the data sets are on the order of a few MB. This fits comfortably on-chip allowing for entirely local computation and avoiding time-consuming off-chip transfers. For the computationally intensive applications, there is massive reuse with each element generally being used at least $O(N^2)$ times. Many large data sets, such as gene databases, are accessed sequentially. In such cases, the data can be streamed onto (and through) the FPGA at Gb rates through dedicated I/O structures.

Use of Simple processing kernels

Many computations are repetitive with relatively simple processing kernels being repeated a very large numbers of times. Examples of this appear in various applications especially those in finding Hidden Markov Models, Bayesian Nets, Dynamic Programming, and various linear algebra computations. For these computations, FPGAs can be configured into a number of processing elements (PEs) specially tuned to the task. PEs therefore compute very efficiently while only requiring small amounts of chip area. The latter can afford a high degree of replication.

Data Flow

Most of the applications listed above are dominated by regular communication: on any iteration, data only need to be passed to adjacent PEs. These short paths minimize communication time.

Associative Computation

Many of the remaining communications are associative operators: broad-cast, match, reduction, and leader election. These occur in the following scenario. Parallel

computations need to be collected to see which yielded the current best solution and a leader needs to be elected. Upon completion of a first set of computations and the computation continues into the next phase, the leader broadcasts a new set of parameters to the rest of the PEs.

Alternatively, a phase ends with data from all the PEs being combined (e.g. summed) to form a solution or to determine status (e.g. done/not done). In all of these cases, FPGAs can be configured to execute the associative operator using long pathways on the chip. The result is that rather than being a bottleneck, these associative operators afford perhaps the greatest speed-up of all.

Other Fine-Grained Parallel Hardware Models

So far we have described non-programmable components. FPGAs can also be configured into stored-program parallel processors. Several standard models are possible, most of which have been well-studied. Three are: (i) SIMD arrays, i.e. a collection of simple PEs that execute programs lock-step on their own data and usually connected via a mesh network; (ii) systolic arrays, similar to SIMD arrays but often having even simpler PEs and used for algorithms with regular data flow; and (iii) fine-grained multicomputers, i.e. collections of bare-bones microprocessors. Each of these is efficient for particular classes of algorithms.

Referring now to FIG. 3, an exemplary template 28 includes a wrapper portion 28a and an application logic portion 28b. In this particular example, a generic template has been defined as a synchronize data pipeline wrapper 28 which includes customized application logic 28b. As will become apparent from the description herein below, the wrapper portion 28a provides predetermined input and output interfaces 28c, 28d which allow the wrapper to accept data from and provide data to data routing elements. The wrapper also includes a predetermined interface to an application specific logic block 28b which customizes the function which the block 28 will perform. Thus, it should be appreciated that a single generic wrapper or template can serve as the basis for many different functional blocks or components

by the addition of particular application specific logic blocks. In a preferred embodiment, the wrapper portion 28a is provided by the system (i.e. is built into the system) and the logic blocks 28b are supplied by a user (e.g. a bioinformatics practitioner or computational support person in a bioinformatics group) for a specific application.

Referring now to FIG. 4, a system 30 for computing a solution to a BCB application includes an input circuit 32 having a user-provided input 34. In this particular example, the user-provided input is in the form of a data vector generated as a result of an experiment. The data vectors are provided to a dot product and sum (DPS) circuit 38 which includes particular DPS logic 40 supplied by a user. Once the DPS circuit performs the desired dot product computations, the results are provided to a covariance and inversion (CAI) circuit 44 which includes particular CAI logic 46 supplied by a user. Once the CAI circuit performs the desired covariance and inversion computations, the results are provided to a regression and correlation (RAC) circuit 50 which includes particular RAC logic 52 supplied by a user. The RAC circuit provides an output to an output circuit 56 which includes user-defined logic for providing correlation results for analysis.

It should be appreciated that dot products are vector operations and thus ae well-suited for operation on the BCB data which is essentially presented in rows (since vectors are also represented as a rows of numbers). It should be appreciated that the it is possible to have vectors of different sizes (e.g. a vector of length 10, length 100, length 1000, etc...). Since the DPS block, among other things, sums up the length of the entire vector, it should be appreciated that more time is needed to sum a vector of length 1000 than is needed to sum a vector of length 100 (i.e. simply because of the difference in the number of data elements).

As will become evident from the description of FIG. 5 below, it is known that dot product operations take a predetermined amount of time and which is presumed to be more time than is required for the operations performed by the down stream CAI block 44. This presumption is made based on knowledge of the size of the data vectors being processed. Furthermore, in any particular application, the particular number of DPS units 38 feeding the

collection network 62 will be selected in accordance with a variety of factors including but not limited to the number of vector elements to be processed (i.e. the number of DPS circuits used in a particular application will vary in accordance with the length of the vectors which are specified). Thus, the amount of time taken to do the dot products is being balanced against the amount of time required to perform the downstream operation (i.e. the operation performed by the CAI circuits). It should also be understood that the amount of time taken to perform the dot product computations will vary according to the application and according to the size of the vector which is specified. It should also be understood that the degree of parallelism which can be achieved in the system and the width of the collection circuit will also vary according to the size of the vector which is specified. It should appreciated that in accordance with the present invention, the number of DPS circuits used in a particular application, the degree of parallelism in the system and the width of the collection circuit is automatically selected by the system based upon user-provided input data.

It should appreciated that many different resources are balanced in this technique including hardware resources (e.g. multipliers, available memory, etc...), units of time (e.g. that computations times in different portions of the system are selected appropriately), the mathematical properties of the Steiner system, and the amount of time required for different mathematical operations.

It should also be appreciated that FIG. 4 illustrates a mathematical or logical problem being solved and that components 32, 38, 44, 50 and 56 are software components (e.g. wrappers) having standard interfaces (represented by arrows 36, 42, 48 and 54 in FIG. 2) to data routing elements which are customized for a particular application via the addition of

application specific logic blocks. The application specific logic 34, 40, 46, 52 and 58 accepts the data from the wrappers and performs certain desired user-defined mathematical functions. Thus it can be seen that with the wrapper approach, the flow of data is managed outside of the application specific logic.

This approach permits re-use of wrapper components 32, 38, 44, 50 and 56 used with different application specific logic components. That is, a user can replace one or all of the application specific logic components 34, 40, 46, 52, 58 with a new set of application specific logic components without interrupting the flow of data within the system since the components 32, 38, 44, 50, 56 can continue to feed data to the new application specific logic components and pull results out of the application specific logic components for further processing.

Referring now to FIG. 5, an aggregate block 60 corresponding to a physical implementation of the DPS and CAI functional blocks described above in conjunction with FIG. 4 and appropriate for use in a field programmable gate array (FPGA) includes a plurality of DPS circuits 38a – 38c. Each of the DPS circuits perform the same functional DPS operation which is specified by DPS logic blocks 40a –40c. Each of the DPS circuits receive an input along one of respective paths 36a –36c and provides an output to a rate balancing collection circuit 62. The collection circuit receives the data provided thereto in parallel along paths 64a – 64c and provides the data along a serial path 42 to a CAI circuit 44.

The CAI circuit is provided from a serial connection unit 44 which includes two pieces of application logic 46a, 46b coupled back to back. Thus, serial connection unit 44 is selected such that it allows some number of pieces of application logic (in this example, two pieces of application logic) to be connected back to back.

It should be appreciated that circuit 60 illustrate three types of connection blocks which can be used to provide an FPGA circuit. Specifically, element 38 illustrates a component having a wrapper with predetermined interfaces and user-defined custom

application logic; element 62 illustrates a collection circuit and element 44 illustrates a serial connection unit. It should be appreciated that these are only a few of several types of standard connection blocks which can be used.

In this exemplary embodiment, it is recognized that certain computations require more time than other computations. In particular, it is recognized that the computations performed by the DPS circuits to perform the DPS function are more time consuming than the computations performed by the CAI circuits to perform the CAI function.

In the example shown in FIG. 5, it is assumed that the DPS computations are slower that the CAI computations by a factor of three. Thus, to avoid a bottleneck of data and to achieve a desired processing speed, the data provided to the DPS circuit is distributed into three paths and fed to three separate DPS circuits 36a – 36c each of which perform the same computation on the data provided thereto.

DPS circuits 36a – 36c provide output data to the collection circuit 62 which appropriately combines the data as necessary and provides the data to the CAI circuit. Since it takes longer to compute each input to the collection circuit 62 than it does for the CAI circuit to performs its computations, three DPS circuit are used at the input to the collection circuit 62 while only one CAI circuit is used at the output of the collection circuit 62. In this manner rate balancing is achieved.

Although in this exemplary embodiment is shown to include three DPS circuit and one CAI circuit, other combinations of circuits can also be used. The particular number of circuits to use on either side of the collection circuit 62 is selected such that the aggregate operating rate of the slower circuits (e.g. the DPS circuits in FIG. 5) is about equal to the operating rate of the faster circuit (e.g. the CAI circuit in FIG. 5).

Referring now to FIG. 6, a portion of an FPGA 70 includes three aggregate blocks 60a –60c each of which is identical to each other and identical to the aggregate block 60 described above in conjunction with FIG. 5. Thus, FIG. 6 includes three instances of the aggregate block 60 described above in conjunction with FIG. 5.

By defining an aggregate block 60, it is possible to determine how many of such aggregate blocks 60a can be used in an FPGA given the resources available in the FPGA. In determining the number of aggregate blocks which can be included in an FPGA, a number of factors are considered including but not limited to:

1. the type of FPGA;
2. the number of available multiplier circuits in the FPGA;
3. the number multiplier circuits required to implement an aggregate block on the FPGA;
4. the amount of available memory in the FPGA;
5. the number of correlation, inversion and regression units (# CIR) which may be computed as:

   (multipliers per chip) / (multipliers per CIR) = # CIR;
6. the dot product sum time (DPS Time) which may be computed as:

   (vector length) × (DPS cycle time) = DPS Time;
7. the ratio of DPS to CIR units (DPS/CIR) which may be computed as:

   (DPS time) / (CIR time) = DPS / CIR;
8. the number of DPS units which may be computed as:

   (DPS / CIR) × (# CIR) = # DPS
9. a mathematical constraint on the number of allowed inputs to the Steiner system as well as the number of allowed outputs from the Steiner system which may be computed as:

$$\max_m \binom{m}{3} \leq (\#\ \text{DPS}) = \#\ \text{X memories};$$

in which m is a range of integers

10. the combinatoric efficiency which may be computed as:

$$\binom{m}{3} / (\#\ \text{DPS}) = \text{combinatoric efficiency};$$

11. the amount of vector time which may be computed as:

min[(DPS time) / (DPS/CIR), CIR time] = vector time; and

12. the result rate which may be computed as:

(# DPS × combinatoric efficiency) / (vector time) = result rate.

13. the existence of any specific purpose circuitry on the FPGA.

It should thus be understood that a variety of different types of specifications of resources available and resources required can be made during a resource balancing process. For example, there are inputs from mathematical requirements, inputs from hardware specifications, inputs from the application data and features of the design as it is implemented in hardware.

It should be appreciated that some of the above factors are related to hardware resources (e.g. the number of available multiplier circuits and the existence of any special purpose circuitry in the FPGA is a function of the FPGA being used) while other factors are related to application specifications (e.g. the vector length, multipliers per CIR, application data length). It should also be appreciated that the input values will affect the design output. Lastly it should be understood that some of the above parameters/characteristic could set limits or drive certain circuit requirements.

With respect to the DPS cycle time, once the user has specified their computation (i.e. via the user-supplied application logic) and some analysis has been done then it is possible to estimate the amount of time required by

the hardware to perform the computation per cycle. For example, in the case of a vector, the user supplied logic may be one or a series of instructions to sum over the length of the vector in which case the cycle time would be the amount of time required to add one summand into the grand total.

It should be appreciated that the number of aggregate blocks which can be included in an FPGA may be different for different FPGAs. It should be appreciated that in accordance with the present invention an arbitrary number of replicas is allowed. Stated differently, the FPGA is queried to determine how many of a particular type of block can be supported and then at compile time, a decision is made as to how many instances of that block should be created.

Also, it should be appreciated that when a compilation (or re-compilation) is done for the same application but with different FPGAs, the same types of blocks are provided (e.g. the same aggregate blocks 60) but different numbers of the blocks may be used depending upon the quantities of resources available on each different FPGA.

Referring now to FIG. 7, a system adapted to rapidly implement one or more bioinformatics algorithms includes a host 82 having an input circuit 84 having user-provided input 86. The user-provided input may, for example, be provided in the form of a data vector generated as a result of an experiment.

The data is provided to an input distribution network 90 which distributes the data to a plurality of the aggregate circuits 60a –60c. Since the amount of data available for computation is relatively large, the distribution network 90 utilizes logic related to the Steiner system to supply data to the aggregate circuits. This approach reduces the amount of data which must be retrieved from input block 84 to supply data to the inputs of the aggregate

It should be appreciated that while the number of different combinations of inputs and outputs which the distribution circuit 90 may have is large, it is not totally arbitrary. There exits a mathematical relationship between the number of inputs and the number of outputs which the Steiner system network 90 may have. Thus, when the system (e.g. compiler 18 in FIGs. 1 and 1A) is selecting the number of inputs and outputs with which to provide the distribution network, it must select a number of inputs and outputs within an allowed range such that the mathematical relationships between different resources (e.g. hardware resources, mathematical properties of the distribution and collection circuits, amount pf time required for all operations, etc...) are all satisfied concurrently.

Aggregate blocks 60a – 60c process the data fed thereto and provide data to an output collection circuit 94 which provides output data to a host 96 having an output block 98 which includes logic for further processing output data provided thereto by the collection circuit.

It should be appreciated that collection circuit 94 maybe implemented using a variety of techniques to collect a large number of results and to provide those results to the output block 98 on host 96. Ideally, it would be desirable to take all of the data received at the collection circuit input ports, appropriately arrange it and provide a single output at the collection circuit output port. This is referred to as a parallel architecture since the collection circuit accepts all of the inputs in parallel from the aggregate blocks 60a – 60c and provides a single output block. Alternatively, however, the collection circuit can be provided such that it accepts all of the parallel inputs from the aggregate blocks 60a – 60c and provides a serial output data string. This is referred to as a serial architecture. Alternatively still, the collection network can accept all of the parallel inputs from the aggregate blocks 60a – 60c and provide a series of partially parallel outputs. This is a hybrid architecture in that it is partially parallel and partially serial.

In a practical circuit using present day manufacturing techniques, a fully parallel collection circuit is not practical and some combination of parallel input and partially-parallel output or serial output may be used. Thus, the particular manner in which collection circuit 94 is implemented in any particular application is selected in accordance with a variety of factors including but not limited to the width of the data path at the collection circuit output port,

It should also be appreciated that the type of distribution circuit and collection circuits will be automatically selected by the software system (e.g. by the tool and not specified by the user) based on the factors listed above and the type of application selected by the user.

Thus, the system will consider all factors provided thereto and select a collection network having either a serial implementation, a parallel implementation or an implementation which is some combination or degree of serial and parallel.

It should also be appreciated that the same logic may be used to provide each distribution and / or collection circuit in the system. Similarly, although only one distribution circuit is shown in FIG. 7, other embodiments may include a plurality of distribution circuits and the same logic may be used to provide each distribution circuit in the system.

Before describing FIGs. 8 and 8A, some introductory concepts are described relating to Certain Optimization Methods with Applications in Microarray Analysis. It was found that microarray analysis involves techniques different from standard statistical methods. This is especially true when microarrays are used to "reverse engineer" cell functions. In this case, learning techniques such as Bayesian networks appear promising. The problem is that the computation of Bayesian nets has exponential complexity.

It was found in accordance with the present invention that one method of computing Bayesian nets that appears to be suited for FPGA implementation is the AI optimization technique of hill-climbing. FIGs. 8 and 8A present an implementation of the AI optimization technique of hill-climbing which involves a novel use of computational cells coupled with associative processing hardware.

Reverse engineering molecular pathways was the original motivation for microarrays. A computational technique that enables progress in this area enables progress in solving one of the fundamental problem in all of biology. Hill-climbing is an important optimization technique with applicability far beyond microarrays. Also, many problems that are solved using other techniques can be recast as hill-climbing problems if there is a good implementation.

Turning now to FIGs. 8 and 8A, a successive approximation control block 102 includes a predefined portion 104 which includes pre-defined input and output interfaces as well an application logic interface. The control block 102 also includes user-supplied application logic blocks 106, 108 and 110.

As explained in conjunction with FIGs. 3-7, control block portion 104 may be provided from a defined control template which is then customized via the addition of particular application logic (e.g. application logic 106, 108, 110) to perform a particular function. In this example, the successive approximation function is achieved by the addition of application logic block 106 which defines a scoring function, application logic 108 which defines an initial state and application logic 108 which defines next state selection.

Fig. 8A illustrates an exemplary implementation 112 of the successive approximation control block 102 of FIG. 8. As shown in FIG. 8A, parallel computations need to be collected to see which yielded the current best solution and a leader needs to be

elected. Upon completion of a first set of computations and the computation continues into the next phase, the leader broadcasts a new set of parameters to the rest of the PEs.

A Bayesian network is defined to consist of: (1) an acyclic digraph G in which nodes represent expression levels of different; (2) genes and edges indicate a direct dependence of expression level of one; (3) gene on another; (4) a set of probability distributions theta consisting of (4a) marginal probabilities for the nodes in the digraph with no parents and (4b) conditional probabilities for the nodes in the digraphs with parents.

Given G and theta, the joint probability of the observed expression levels for the nodes in G can be calculated under the additional Markov assumption that the expression level of any node in G is independent of nodes which are not descended from it. Under certain simplifying assumptions regarding the prior probability distribution of the pair (G, theta), a computationally simple formula can be derived for the posterior probability of G given a set of observed expression levels.

-- Parameter set = all possible digraphs G
-- Scoring function = posterior probability of G
-- Sample data is the set D of observed expression levels

Computational Methods

A scoring function can be computed in parallel with each processor mapped to a node in G. A local search technique such as greedy hill climbing can be used with a large number of independent realizations of G. Local changes to G consist of: (1) operations such as edge addition/deletion/reversal; (2) another local search-based method; each node is given a copy of G through broadcast; (3) each node varies G in a different way (looks in a different direction) in parallel; (4) the best result is computed through leader election; (5) a new graph is again distributed to array and the process is repeated.

Another method of parallelism would be to try different parts of the space as seeds in parallel.

Although only a few exemplary embodiments of this invention have been described in detail above, those skilled in the art will readily appreciate that many modifications are possible in the exemplary embodiments without materially departing from the novel teachings and advantages of this invention.

For example, the systems and techniques described above in conjunction with FIGs. 1-8A could be used in Protein Docking Computations. One of the most important problems in computational chemistry and molecular dynamics with applications in intelligent drug design is finding how and where molecules dock with proteins. The basic science is well understood: the issue is simply one of computational complexity.

Use of direct correlation, rotation through coordinate axis transformation, coordinate axis transformation through indexing. Use of small-integer scoring functions, possibly scoring different features (e.g. core collisions versus surface interactions) separately using saturating counters. Sparse voxel models are typical. No use has been made yet of sparseness. May be a candidate for acceleration techniques Work in progress on using compressed data/hardware structures for the above computations.

The systems and techniques described above in conjunction with FIGs. 1-8A could also be used in Detection of Tandem Repeats and Palindromes. Genomics has created a vast database of sequences. One of the most useful features is subsequences that repeat some number of times (Tandem Repeats) or repeat backwards (Palindromes). Methods of interest involve both precise matches and matches with a small number of errors. Problems are computational complexity and creating flexible implementations that satisfy the large number of variations of these problems.

The computational requirement is somewhat different than that of partial string matching. In the latter, a large number of replacements and insertions/deletions are possible, indicating DP methods. Here, direct use of cascaded comparators is preferable. Our implementation has novelty at the circuit design level. Analysis should proceed faster than the 10G letters/second making this ideal for use as part of a high-end database analysis system.

Accordingly, all such modifications are intended to be included within the scope of this invention as defined in the following claims. It should further be noted that any patents, applications and publications referred to herein are incorporated by reference in their entirety.

What is claimed is:

CLAIMS

1. A system for providing a Field Programmable Gate Array (FPGA) for use in computation of a bioinformatics application comprising:

a first storage location having stored therein one or more sets of domain specific rules for user-defined, custom applications;

a second storage location having stored therein one or more sets of hardware contexts;

a third storage location having stored therein one or more sets of application independent code.

a compiler coupled to receive data from said first, second and third storage locations, said compiler for receiving user-supplied input data, at least one set of domain specific roles, at least one set of hardware contexts and at least one set of application independent code and for providing host interface code appropriate for further processing to provide a host interface program and FPGA code appropriate for further processing to provide FPGA circuit code.

2. A method for rate balancing in a BCB application, the method comprising:

determining a time required for a first circuit to perform a first operation used in the BCB application;

determining a time required for a second circuit to perform a second operation used in the BC application;

determining which of the first and second circuits requires more time to perform its respective operations;

determining how many times the circuit which requires more time to perform its operations must be replicated such that the combination of all such replicated circuits processes data at a rate substantially equal to the rate of the other circuit.

3. A method automatically selecting the amount of parallelism to use in a field programmable gate array (FPGA) comprising:

identifying operations to be performed to process data in the BCB application;

identifying circuits to perform each of the operations;

identifying at least one circuit to be replicated in the FPGA base upon the amount of time required for the circuit to perform its function compared with the amount of time required for other circuits to perform each of their respective functions;

determining the amount of resources available in the FPGA; and

replicating the circuit the number of times required to process data at a predetermined rate without exceeding the resources available on the FPGA.

ABSTRACT OF THE DISCLOSURE

A method and apparatus for providing a programmable device for use in bioinformatics and computational biology (BCB) applications.

5

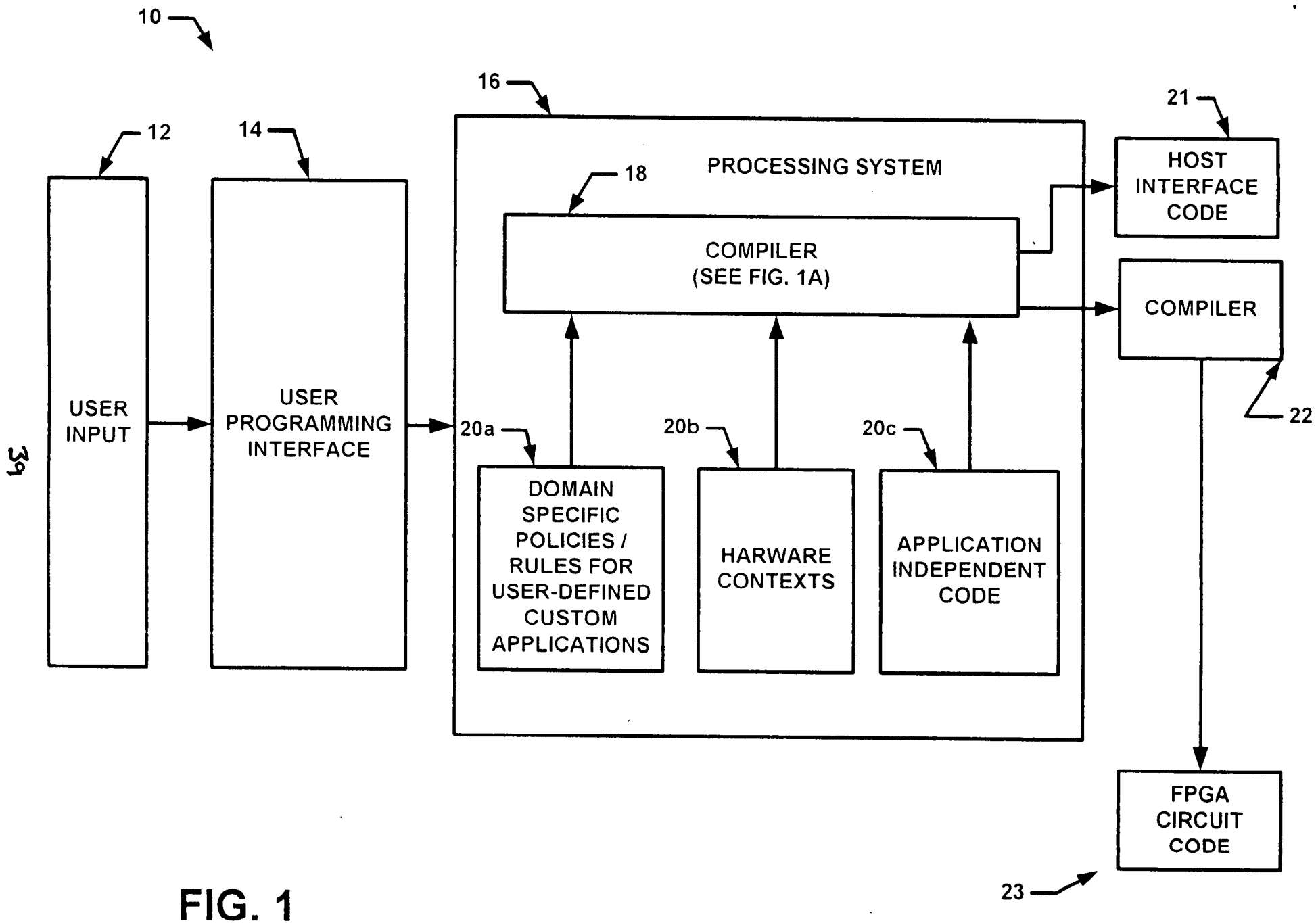

10
16
21
12
14
PROCESSING SYSTEM
18
HOST INTERFACE CODE
COMPILER (SEE FIG. 1A)
COMPILER
USER INPUT
USER PROGRAMMING INTERFACE
22
20a
20b
20c
DOMAIN SPECIFIC POLICIES / RULES FOR USER-DEFINED CUSTOM APPLICATIONS
HARWARE CONTEXTS
APPLICATION INDEPENDENT CODE
FPGA CIRCUIT CODE
23

FIG. 1

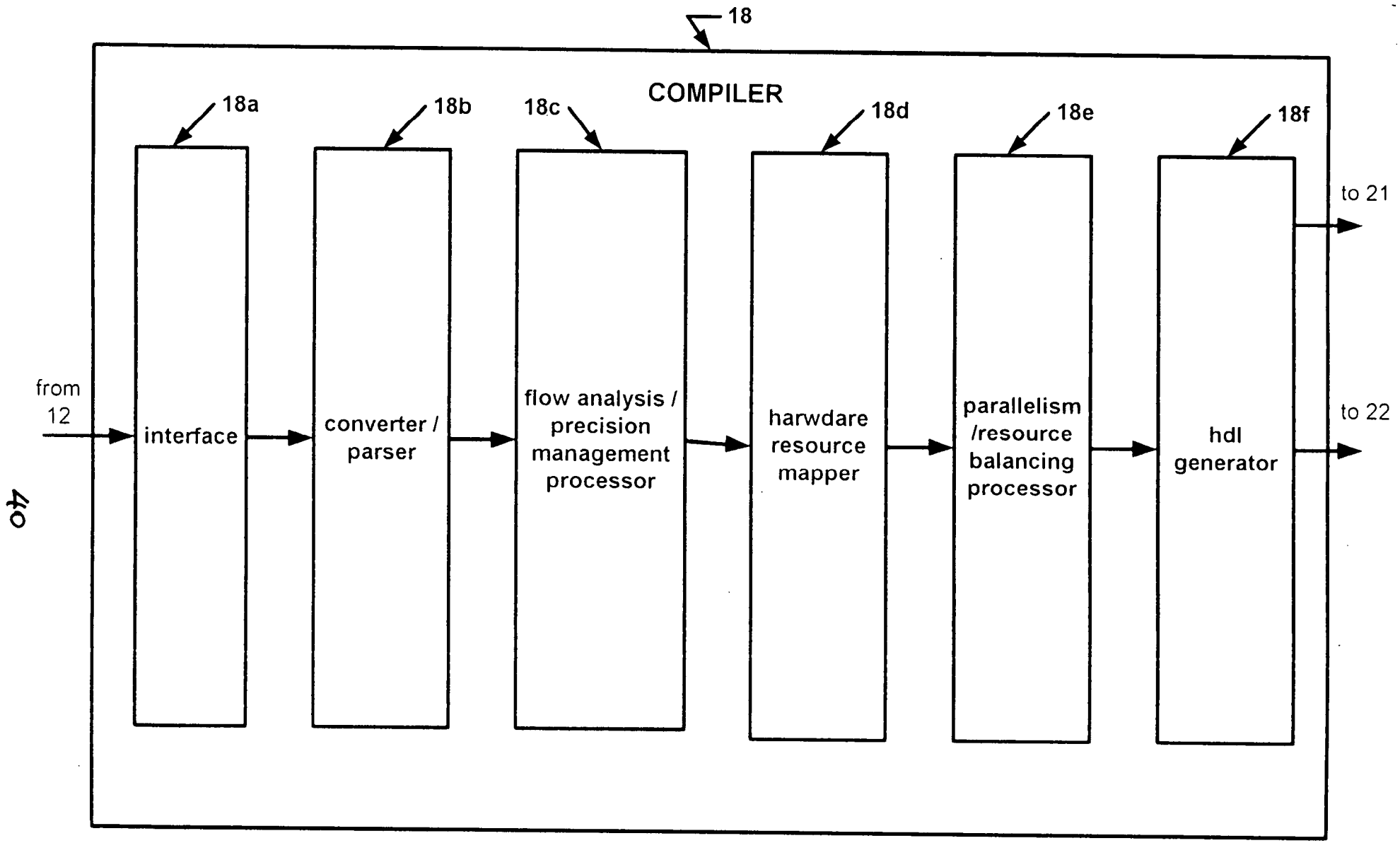

18
COMPILER
18a
18b
18c
18d
18e
18f
from 12
interface
converter / parser
flow analysis / precision management processor
harwdare resource mapper
parallelism /resource balancing processor
hdl generator
to 21
to 22
40

FIG. 1A

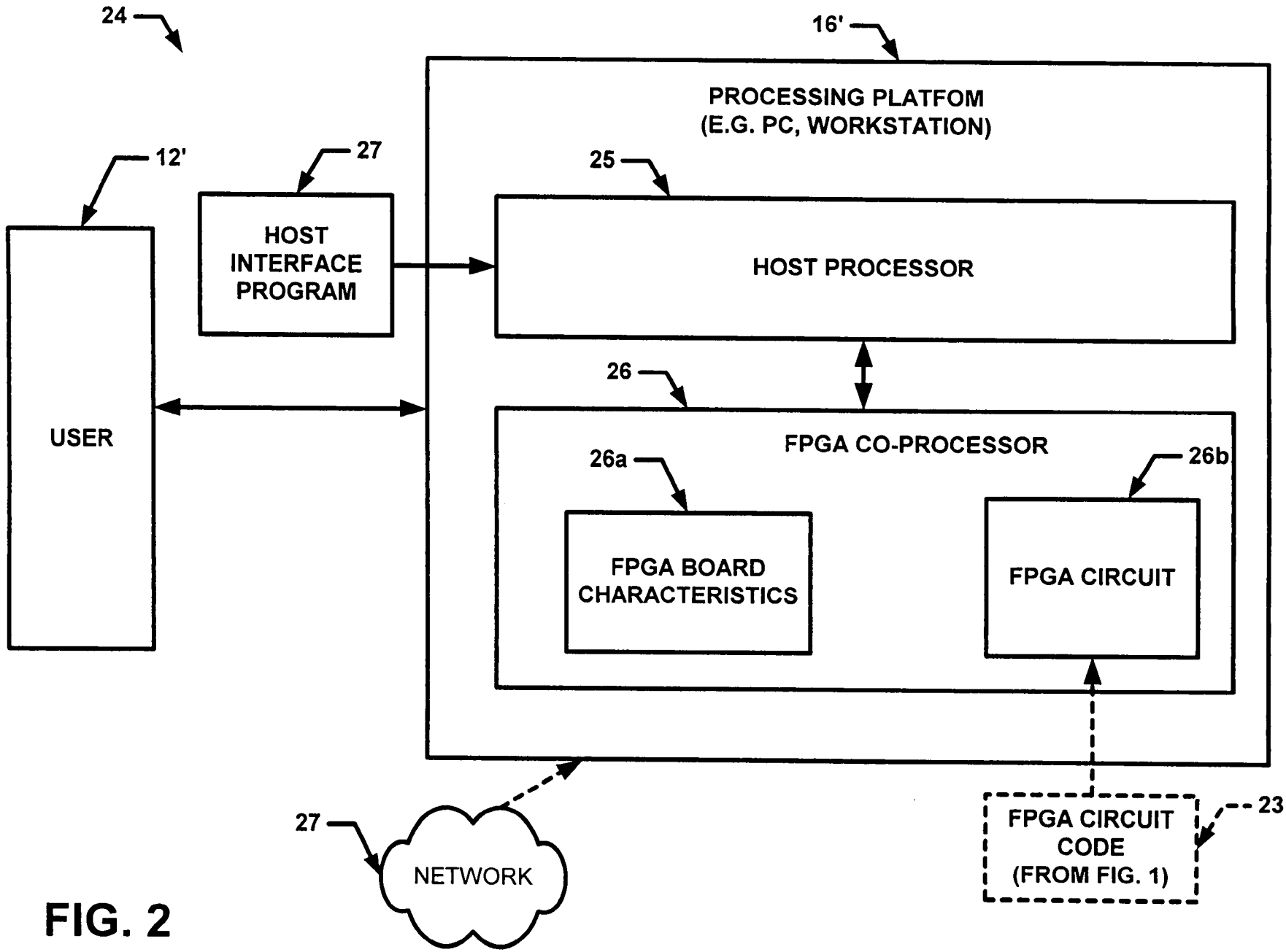

24
16'
PROCESSING PLATFOM
(E.G. PC, WORKSTATION)
12'
27
25
HOST
INTERFACE
PROGRAM
HOST PROCESSOR
USER
26
FPGA CO-PROCESSOR
26a
26b
FPGA BOARD
CHARACTERISTICS
FPGA CIRCUIT
27
NETWORK
FPGA CIRCUIT
CODE
(FROM FIG. 1)
23

FIG. 2

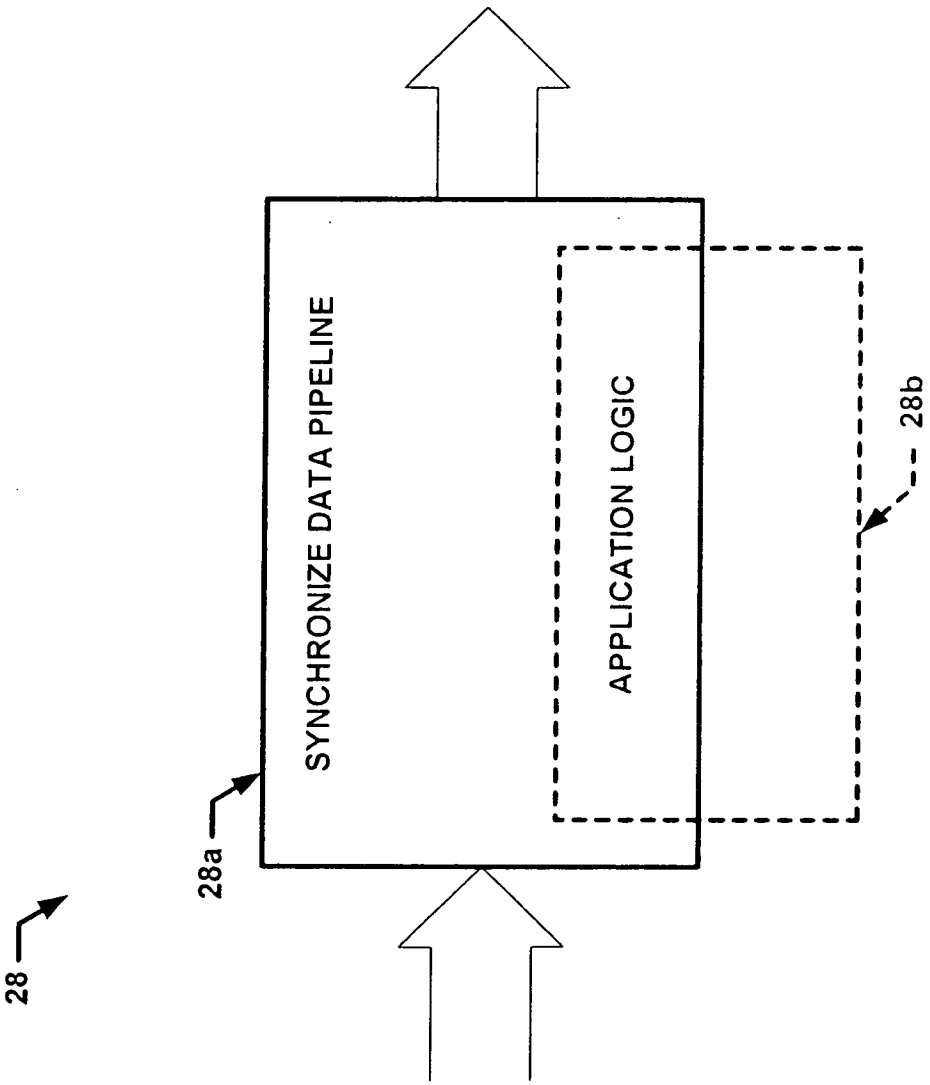


FIG. 3

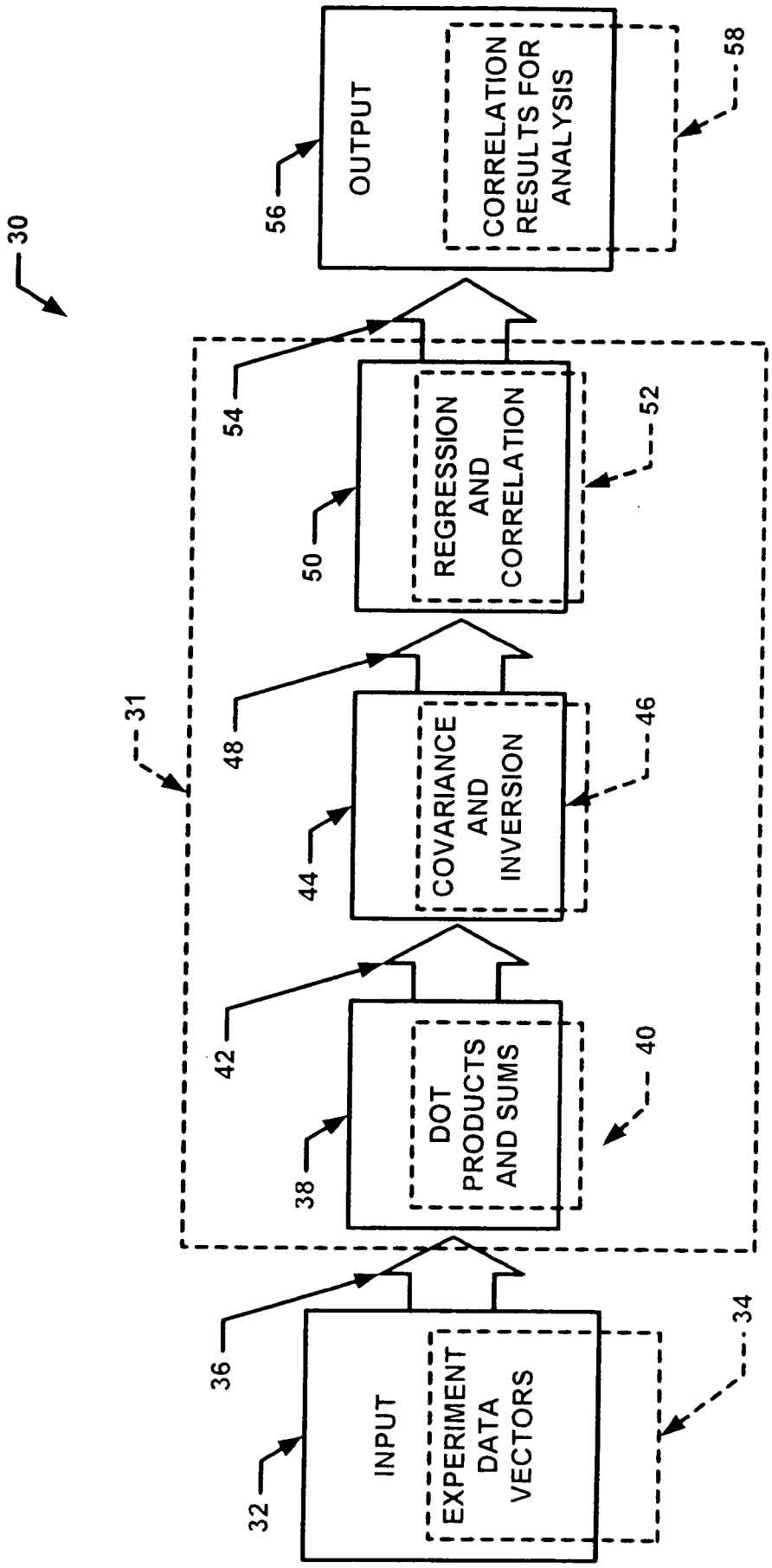

30
31
INPUT
EXPERIMENT DATA VECTORS
32
34
36
DOT PRODUCTS AND SUMS
38
40
42
COVARIANCE AND INVERSION
44
46
48
REGRESSION AND CORRELATION
50
52
54
OUTPUT
56
CORRELATION RESULTS FOR ANALYSIS
58

FIG. 4

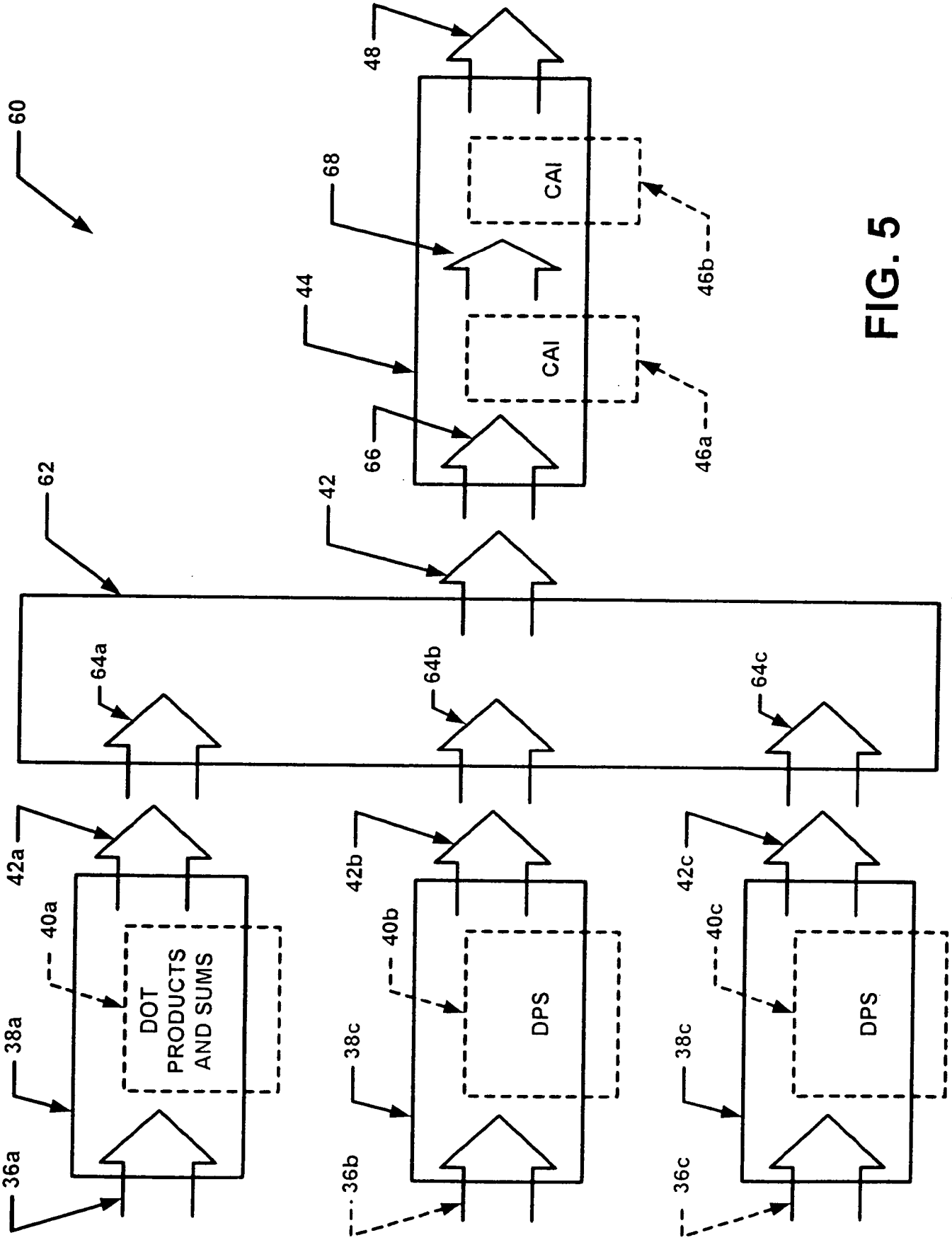
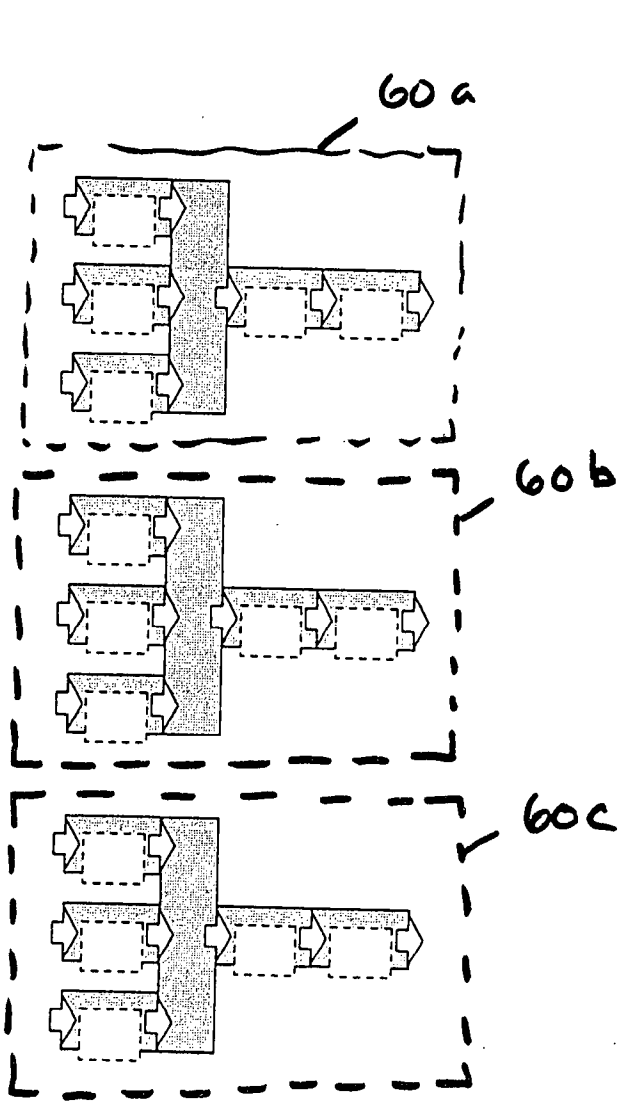

60
48
68
CAI
46b
44
CAI
46a
66
42
62
64a
64b
64c
42a
40a
DOT
PRODUCTS
AND SUMS
38a
36a
42b
40b
DPS
38c
36b
42c
40c
DPS
38c
36c

FIG. 5



45

Figure 6

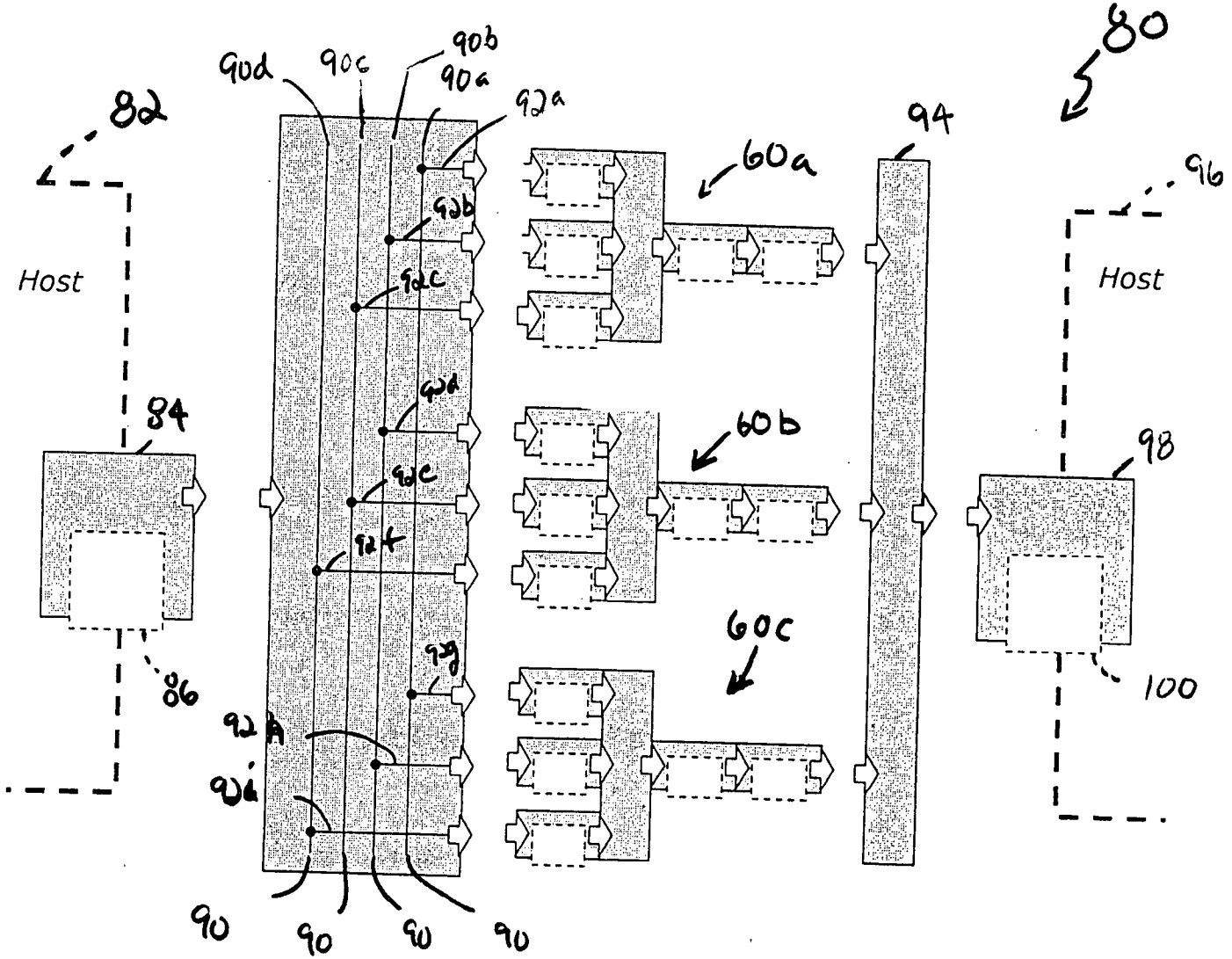

80
82
Host
84
86
90d
90c
90b
90a
92a
92b
92c
92d
92e
92f
92g
92h
92i
90
90
90
90
60a
60b
60c
94
96
Host
98
100

Figure 7

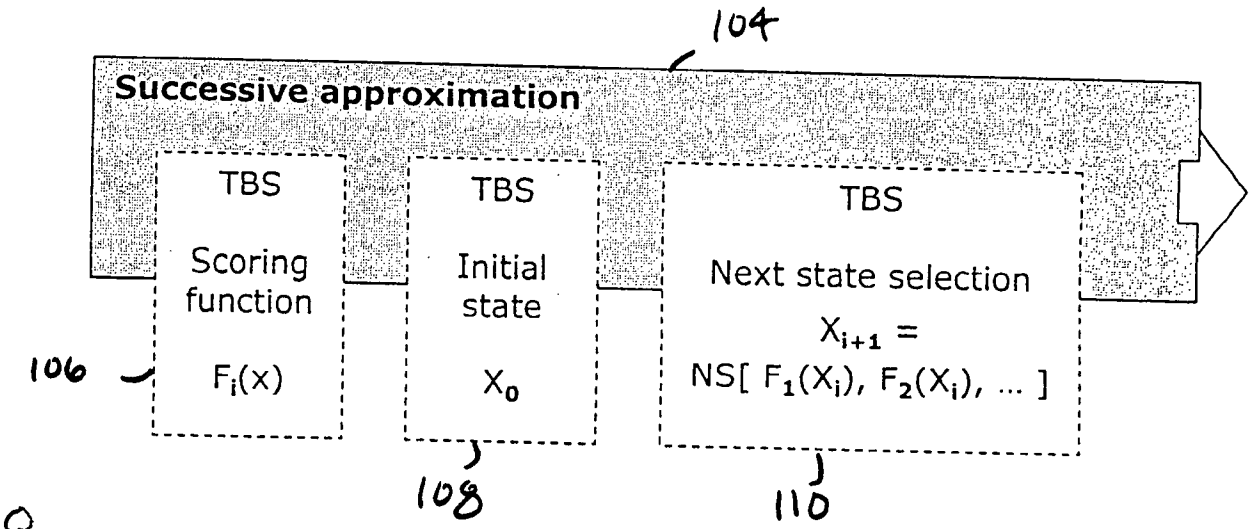


Figure 8

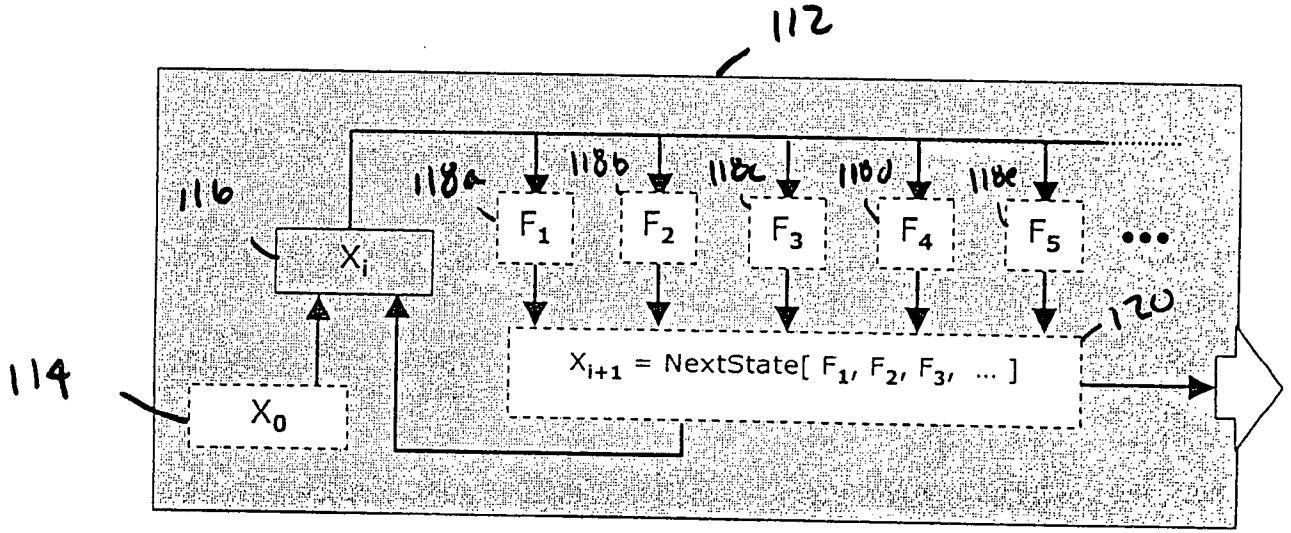


Figure 8A

Research Plan

1 Specific Aims

The need for more cost-effective, flexible, and convenient high-performance computing is a common thread across diverse areas of Bioinfomatics and Computational Biology (BCB) [20, 24, 53, 54]. Some particularly problematic domains are protein structure prediction, phylogenetic analysis, molecular modeling, and modeling genetic networks [7, 11, 41, 48, 57, 58]. It appears that this need will continue to increase: e.g., one trend is the use of what were formerly considered complex computations as components of iterative or hierarchical solutions [22, 45, 52].

The broad long-term objective of the proposed work is to address this problem by augmenting PCs with low-cost FPGA-based computational coprocessors; that is, with plug-in boards similar in cost, use, and ease of installation to a graphics card. The key, however, is not the hardware itself (whose technology is mainstream [3, 5, 44]); rather it is to supply with this *accelerator* a software environment that enables its efficient use. This environment will support a wide range of user sophistication by presenting a variety of interface modes, all of which completely hide the underlying hardware not only from the user, but from most developers as well:

1. Canned applications (e.g., BLAST) for users with little computer programming background,
2. Application/Function/Template libraries and accompanying graphical structures for users with scripting or spread-sheet programming capability; this is for customizing existing applications and creating new ones within established domains, and
3. A high-level language for application programmers to create applications from scratch, or to augment the template libraries to create new families of applications.

The projected deliverable is a factor of 100 to 1000 speed-up over a PC for a wide variety of applications together with an environment for the creation of new applications by ordinary programmers. Since only commodity parts need to be used, the materials and manufacturing cost should be no more than that of the PC itself. We emphasize that this solution is flexible and therefore capable of keeping up with evolving algorithmic designs, not brittle as were some ASIC-based custom computing solutions tried in the past.

Although we have thus far emphasized desktop PC computing, The Proposed System (TPS) is equally applicable to clusters of PCs. In that case, the goal is to use the accelerators to build the computational equivalent of a cluster of clusters: these can then be used to address true Grand Challenge computations, e.g. in molecular dynamics. The potential impact of TPS is therefore (at least) two-fold. First is the creation of knowledge through brute computational capability. Second is the empowerment of the individual researcher: having the equivalent of a supercomputer on the desktop reduces the compute time of "heroic" computations from weeks to minutes. This has further consequences as algorithms (or versions of algorithms) can now be used that would previously have been prohibitively expensive, resulting in improved sensitivity and enabling entirely new applications.

Much of the specific work proposed involves answering two questions: (i) can an FPGA accelerator effect the proposed speed-ups across a number BCB applications? and (ii) can a software environment be created that allows users, with no knowledge of the underlying hardware, to develop new applications? To answer (i) we propose to create a new BCB/FPGA applications, particularly in modeling molecular interactions. To answer (ii) we propose to build and test a prototype software system. Performing this work will require the application of principles from various fields:

- Algorithmics. Good performance on an FPGA invariably requires algorithm reformulation.
- Computer architecture. Determining how circuit and software resources can best be organized and applied.

- Software engineering, to design and implement TPS
- Circuit design, to create HDL code for some template functions for TPS and at least part of the new BCB/FPGA applications.

2 Background and Significance

2.1 Significance

The Human Genome Project and related work have transformed Biology from a purely laboratory-based science to an information science as well [39]. Moreover, this successful execution of Big Science has enabled "the key to scientific progress[, which] is to unlock the brain-power of the individual researcher [40]." This manifests itself, e.g., in the capability of scientists, from their desktops, to access huge databases and to there conduct rudimentary processing of that data. This empowerment has not extended nearly as much to computation: although much can be done with a desk-top PC, there are also severe limitations. The cost of these "beyond-PC" computations in terms of system acquisition, learning curve, maintenance, and convenience has limited "Big Computation" to large research projects.

We propose the logical next step: to **bring to the individual researcher a capability in computation analogous to the existing capability in data access in terms of speed, cost, convenience, and flexibility.** The advantages of improved speed, cost, and convenience are obvious; flexibility less so. Critical is that TPS will not only allow many current large-scale applications to be run at the desk-top: it will also break the chicken-and-egg phenomenon whereby most application developers avoid computationally complex algorithms because the appropriate hardware is out of reach and low-cost accelerators are not developed because there are not enough applications to make them cost-effective. Putting accelerators *within* reach of individual researchers (in terms of both product cost and learning curve) will allow the community of developers to apply known algorithms with complexity greater than, say, $O(N^2)$, often the nominal limit for everyday PC computations with the expected data sets. Perhaps more importantly, it will also inspire the creation of entirely new algorithms and derivative applications. A further advantage of flexibility is derived from the fact that BCB does not occur in a vacuum: the norm is to develop and run computations in close collaboration with experimentalists. Essential to the functioning of this development loop is agile high-performance computing such as proposed here.

We have already verified several BCB applications as amenable to substantial acceleration by TPS (see Section 3). As we are in the exploratory phase of this study, we are still identifying the promising targets: This will continue in consultation with faculty of the Bioinformatics Program at Boston University. Some likely candidates are tasks that are obviously compute bound, including: protein docking [13], motif finding [6, 35], multiple alignment [20, 27, 43], and structure prediction [7, 48]. More broadly, the potential impact of general speed-ups as we have described could reach far beyond research environments: TPS could cost-effectively provide computation wherever a large cluster would be of use. Although quite speculative, one could well envision TPS as part a process for delivering individualized genome-based treatment [1] in hospitals even doctor's offices.

2.2 Other Work

2.2.1 Non-FPGA Based

1. PCs — PCs are cheap, distributed, and flexible. There is much free and commercial software and PCs have a familiar programming environment. It is likely that the vast majority of BCB application executions occur on PCs. The drawback is computational power: it is common that algorithms of up to only $O(N^2)$ complexity (for common data sets) are run. This means that PC applications make liberal use of assumptions and heuristics to limit computational complexity. Also, the complex higher order computations described above are completely out of reach for PCs.

2. PC clusters — PC clusters are popular and becoming more so with many thousands in active use. The performance is up to P-times better than a PC for P processors. For small clusters (fewer than 16 nodes), the material cost can be slightly less than P-times that of a PC. Larger clusters, however, e.g. those with more than 64 nodes, have special housing, network, power, and cooling requirements. In all cases, there is a significant system management cost. There is much less available software than for PCs and specialized programming skills are required to get good performance.

3. Large centralized facilities — These come in many varieties and include the supercomputers and MPPs, e.g. at the national centers, and the huge workstation farms at some companies [2, 14, 18, 50, 53]. As with PC clusters, performance is up to P-times better for P processors. However, it is more likely that this performance scaling will extend to larger problems. Also, the programming model may be simpler. However, the cost is very high, both in acquisition and system maintenance, and the convenience often low. Researchers must apply for supercomputer time, scalability studies are often required, programs must be submitted batch with substantial turnaround times, and rarely is the user permitted more than a small fraction of the resource.

4. ASIC solutions — Application Specific Integrated Circuits (ASICs) are sometimes capable of tremendous performance for certain applications [4, 9, 17, 42]. It is almost always the case, however, that they have limited applicability. Also, they are unable to take advantage of the continuous improvements in process technology.

5. Programmable Non-Commodity Processors — Chief among these have been massively parallel fine-grained SIMD arrays which have periodically had substantial success [8, 10, 25]. The problem here is also in taking advantage of improvements in process technology. Two things makes them promising again, however: advances in FPGA technology (e.g., an entire array can now fit on a single FPGA) and the recent solution (by the PI) of one of the basic control problems that formerly hindered the development of scalable SIMD systems [31].

2.2.2 FGPA Based

What is an FPGA?

Field programmable gate arrays (FPGAs) are commodity integrated circuits whose (apparent) circuitry can be determined, or *programmed*, in the field. This is in contrast to ASICs whose circuitry is fixed at fabrication time. The tradeoff is that FPGAs are less dense and fast than ASICs; often, however, the flexibility more than makes up for these drawbacks. Also, for the right applications, FPGAs obtain speed-ups over microprocessors of from several hundred to a thousand or more. Beyond the configurable substrate, high-end FPGAs also contain embedded ASIC modules, including entire microprocessors. To summarize, high-end FPGAs have the following characteristics.

- Programmable in milliseconds by uploading the desired configuration. This also means that the FPGA can be *reprogrammed* for other applications just as quickly.
- 4 million+ configurable gate-equivalents
- Millions of programmable communication paths, both local and global
- Circuits are generally designed using hardware description languages (VHDL, Verilog); much available Electronic Design Automation (EDA) support
- Design modules often available as "intellectual property" (IP) blocks
- Gigabit interfaces (Infiniband, Gigabit Ethernet, etc.) to off-chip devices
- Hardwired on-chip microprocessors, busses, memory modules, and multipliers.

FPGAs have yet another advantage: currently, FPGA chip development *is driving process technology*, a role held until recently by DRAM chips [12]. As a result, TPS has the critical attribute for new IT products: the ability to "ride the technology curve" as stated by Moore's Law.[1] As new generations

[1]Transistor density on a chip doubles every 1-2 years

of process technology—and thus FPGAs—emerge, existing *hardware* designs can be ported to them in much the same way that existing software can be loaded onto a new faster computer, thereby obtaining an analogous boost in performance.

FPGA Products and Technology

1. Turn-Key Systems — Turn-key systems are available from TimeLogic [55]. These consist of clusters with dedicated accelerators whose hardware is similar to what we are proposing to use here. For the set of programs that is supported—BLAST, HMMer, and Smith-Waterman—very high speed-ups are achieved. The drawbacks are cost and flexibility. Minimum configurations start at $70,000 and typical systems cost $500,000 and up. Perhaps even more importantly, these systems can *only* run those applications supplied by the vendor.

2. General Purpose EDA Tools — General purpose FPGA boards [3, 5, 44], can yield tremendous speed-ups when configured appropriately. The problem (addressed here) is that configuration for high-performance applications requires skilled application analysis, algorithm development, and circuit design. Much preferred would be that an application programmer or end-user could obtain similar results. Much work has been in done in developing Electronic Design Automation (EDA) software toward this goal:

- **IP.** Because circuits are created in software before they are cast into an ASIC or configured into an FPGA, it is trivial to reuse the design. Designs, called Intellectual Property (IP) blocks can therefore be sold. Most often, IP blocks are nitty-gritty system components such as bus, I/O, and memory interfaces, or embedded microcontrollers.
- **Hardware Design Language (HDL) code from High-level language (HLL) code.** Examples are Handel-C [15] and Forge [59]. These have proven extremely useful in design management, coupling simulation and design, and hardware/software codesign. However, they are still primarily intended for circuit designers rather than HLL programmers.
- **Domain Specific Environment.** System Generator for DSP [60] allows users to design circuits via a data flow interface; i.e., by specifying how signals flow through various components such as converters, filters, and transformers. The drawback of this system is that it currently only works for DSP. Creating such a system for BCB is one of the goals of this project.
- **Domain Specific HDL code from HLL code.** HLL to HDL has met with some success when the application of the end-product is constrained. An example is the SA-C language and compiler developed for computer vision processing hardware [47]. Constraining the domain allows the HLL to be restricted, making the problem of converting HLL to efficient HDL tractable. The drawback is that new language constraints need to be developed for BCB. Creating such a language and compiler is one of the goals of this project.

2.2.3 Summary of Previous Work

Of the existing technologies, only large-scale MPPs/Supercomputers and dedicated hardware provide high performance. MPPs/Supercomputers are extremely expensive, inconvenient to use, and not easy to program. Dedicated hardware is expensive and not programmable except by the vendor. The ideal solution, of course, combines the best of all of these: the low cost, convenience, and programmability of the PC with the high performance of the MPP/Supercomputer or dedicated hardware. We believe that this combination can be achieved by creating systems for BCB that are analogous to SA-C for Computer Vision and System Generator for DSP.

2.3 Solution Requirements – Getting from specific work to long-term goal

Critical to explaining the significance of the proposed work is the series of transformations that are required to get from problem formulation to program execution. See Figure 1 (after [29, 36, 51]). The typical program development flow for PCs is shown on the right, clusters in the middle. Note that a

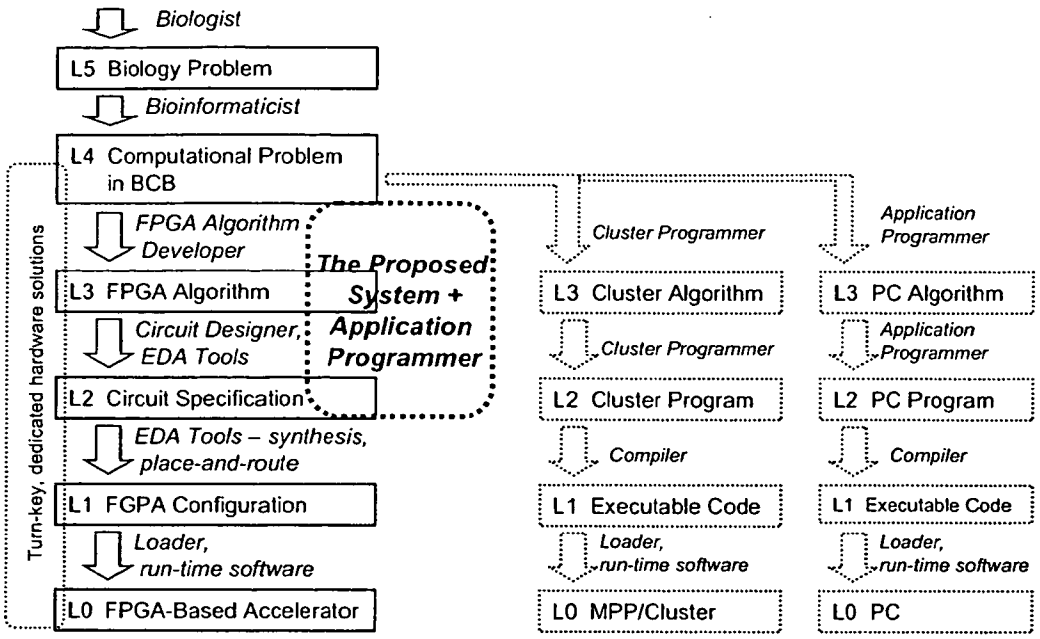


Figure 1: Transformations required to get from problem to running computation.

parallel computer often requires not only a modified program, but often *also a different algorithm from a PC* [19, 23, 36, 51]. This, in turn, generally requires the additional set of skills which distinguishes the parallel programmer from the everyday application programmer. The distinction is even greater in the FPGA accelerator development flow. In particular, two of the transformations require skills not found in either application or parallel programmer: computational problem to FPGA algorithm and FPGA algorithm to circuit specification.

Recall that there are two broad goals to this work: demonstrating FPGA applications for BCB and that TPS is viable. The first of these involves working with Bioinformaticists to isolate computational problems in BCB (at L4), creating FPGA algorithms (L4 to L3), and demonstrating their speed-up (L3 to L0, but with L2 to L0 fully automated with off-the-shelf software). The second involves creating a software system to make the FPGA development flow appear to the application programmer to be nearly indistinguishable from the PC development flow.

Again, we emphasize that a novel software system is required to perform this function. Algorithms that yield optimal performance on an FPGA are almost always substantially different from algorithms that provide the corresponding function on a PC or cluster (see Section 3 for numerous examples). Moreover, the algorithm characteristics that give BCB applications their speed-ups on FPGAs (described in the next section) are generally not even "in the toolbox" of the typically PC or cluster programmer. This leads to two issues that are fundamental to TPS. The first is that software constructs or functions must be provided to the programmer to enable the use of those FPGA characteristics that give FPGAs such power in executing BCB applications. The second is more fundamental: FPGA applications *must use these constructs in order to obtain maximal performance.*

3 Preliminary Studies

This section is divided into two parts following the two major aspects of this study: Appropriateness of BCB applications for FPGA acceleration and the viability of TPS.

3.1 FPGAs for BCB

There are three parts to this subsection: architectural analysis, case studies, and EDA enhancements. The last of these involves automating the L3 to L2 transformation.

3.1.1 High-Level Architectural Analysis

The (related) goals of high-level architectural analysis are to find: (i) whether there is a broad match between overall requirements of a computation with overall capabilities of a technology, and (ii) whether there are recurring structures within a computation that can be implemented particularly efficiently in a technology.

Characteristics of BCB Computations

We have analyzed a number of applications, but for brevity, describe only two: genomics through string processing and functional genomics through microarray analysis. We then summarize the application characteristics.

In genomics, the data structures are character strings, the algorithms process those strings. Many algorithms fall into a relatively small number of classes: exact and inexact string matching using classical techniques, partial string matching using dynamic programming, string structure analysis using Markov models, and string comparisons and analysis using suffix tree techniques. Data set characteristics are as follows: they have a small alphabet, from 4 to 20 letters; there are from 10s to billions of characters per string; there are from 1s to millions of strings; the data are noisy; and comparisons often require evaluation with a match table with size 20 x 20 x score. The computational complexity of some sample algorithms is as follows. For n characters and k strings: simple matching using suffix trees is $O(kn)$, partial matching using dynamic programming is $O(n^k)$, while assembly can be $O(exponential)$ in the number of fragments.

Moving on to functional genomics, one popular technique involves using microarrays to measure simultaneously the expression products of thousands of genes in a tissue sample. The individual data points, however, are very unreliable. A microarray study consists typically of at most 100s of microarrays, e.g. from patients or experiments. Here we process m vectors of size n and corresponding to m genes and n tissue samples. In microarray data processing, the correspondence of computation and biology is not as direct in microarray analysis as it is in sequence analysis and a wide variety of techniques are used. Some classes of algorithms that are used are as follows: clustering, dendrograms, applications of standard statistics after preprocessing, self-organizing maps, Bayesian nets, and AI inspired machine learning. The data sets are remarkably uniform: expressions require 2 to 4 bits, outcomes = 1 bit; there are 10s of thousands of genes and up to 100's of samples; the data are very noisy with some data missing altogether. Again, there is a wide variety of computational complexity from $O(n)$ to $O(exponential)$.

Some characteristics we can derive from these and other BCB applications are as follows.

1. Low resolution data — Data elements often require only few bits. This is true of sequence data, microarray data, and even molecular dynamics if the problem is cast the right way.
2. Large but manageable data sets. There are thousands of genes, thousands of samples (at a time). When the data sets are very large, the data are accessed sequentially such as when streaming through a database. Other computations are inherently I/O bound.
3. High-dimensional parameter sets that must be searched or enumerated to identify a solution. Many preferred algorithms have very high computational complexity.
4. Simple performance criteria (score functions) that are evaluated for each candidate parameter vector. While the algorithms may have very high computational complexity, the individual computations tend to be quite simple.
5. Decomposable search strategy. Many algorithms are easily partitioned into multiple independent problems.

Using FPGAs for BCB

Here we give a high-level argument why this speed-up trend is likely to continue through a number of BCB applications we have not yet examined in detail. In other words, where does the consistent speed-up by a factor of 100 to 1000 (claimed in the Overview) come from? In computer architecture, good performance is dictated by fast cycle times, but just as important are parallelism and locality (the latter being closely related to cycle time). In other words: How many processing units can we simultaneously bring to bear and how long does it take to get the data to them? It is often the case that the goals of maximizing parallelism and locality are at odds with one another; in fact, dealing with this problem is at the heart of the field of parallel computer architecture. However, *in the mapping of BCB to FPGAs parallelism and locality can often be achieved together.* Some of the reasons are as follows.

Data elements have a small number of bits — Since only a few bits need to be processed per datum, we can configure the gates of the FPGA into many thousands of Arithmetic Logic Units (ALUs). This is in contrast to PCs where there are a small, fixed, number of large ALUs. The latest FPGAs also have optimized dedicated circuitry to support hundreds of parallel, moderate precision, multiplications.

Data set sizes and management — For many applications, the data sets are on the order of a few MBs. This fits comfortably on-chip, allowing for entirely local computation by avoiding time-consuming off-chip transfers. For the computationally intensive applications that are our target, there is massive reuse with each element generally being used at least $O(N^2)$ times. Many large data sets, such as gene databases, are accessed sequentially: in that case the data can be streamed onto (and through) the FPGA at Gb rates by using the dedicated I/O structures.

Simple processing kernels — Many computations are repetitive with relatively simple processing kernels being repeated very large numbers of times. For these computations, FPGAs can be configured into a number of processing elements (PEs) specially tuned to the task. PEs therefore compute very efficiently while only requiring small amounts of chip area. The latter can afford a high degree of replication.

Dataflow — Most of the applications listed above are dominated by regular, local, communication: on any iteration, data only need to be passed to adjacent PEs. These short paths minimize communication time.

Associative Computation — Many of the remaining communications emerge from the application of associative operators: broadcast, match, reduction, and leader election. A detailed example of this is given in the "successive approximation" case study. In all of these cases, FPGAs can be configured to execute the associative operator using the long communication pathways on the chip. The result is that rather than being a bottleneck, these associative operators afford perhaps the greatest speed-up of all: *processing at the speed of electrical transmissions.*

Other Fine-Grained Parallel Hardware Models — So far we have described configuring programmable circuits into non-programmable components. FPGAs can also be configured into stored-program parallel processors. Several models are possible, most of which have been well-studied. Three are: (i) SIMD arrays, i.e., a collection of simple PEs that all execute the same program in lock-step but on their own data and that are usually connected via a mesh network; (ii) systolic arrays, similar to SIMD arrays but often having even simpler PEs and used for algorithms with regular dataflow; and (iii) fine-grained multicomputers, i.e. collections of bare-bones microprocessors again connected via a regular local network. Each of these is efficient for particular classes of algorithms.

3.1.2 Using FPGAs for BCB: Case Studies

Case studies other than by PI's group

The utility of FPGAs for some BCB computations has been demonstrated. Existing products from TimeLogic have achieved speed-ups in the 100s for BLAST, HMMer, and Smith-Waterman [55]. Other

academic studies have also shown great speed-ups, but for drastically simplified problem formulations [26, 61, 62].

A Microarray Analysis Application

Kim, et al.[38] proposed the following problem: what is the set of three genes whose expression can be used to determine whether liver tissue samples are metastatic or non-metastatic. They further proposed that use of linear regression would be appropriate to evaluate the gene subsets over the available samples. Since there are thousands of potential genes, 10^{11} to 10^{12} data subsets need to be processed. Although simple to implement, he reported that this computation was intractable even on his small cluster of PCs. Our serial implementation concurred: for roughly 100 samples, 10,000 genes, and 13us per iteration, the computation would have taken nearly three weeks on a 1.7G Pentium 4. Our FPGA implementation resulted in a speed-up of a factor of more than 1500, resulting in a compute time of 10 minutes [56].

The computation of each set requires three steps: dot products and sums, covariance and inversion, and regression and correlation. To take advantage of the parallelism available, units were maximally replicated. To take advantage of the massive data reuse, we constructed a vector store and distribution unit based on a Steiner system [28]. Other notable features of the implementation include the handling of missing data, speed-matching the components, and the precision management.

Besides the speed-up and the techniques invented as part of this solution, the significance of this case study also lies in the ease with which it can be extended to many other microarray data analysis applications. In the next section we describe an extension discriminant analysis [46].

Sequence processing

We have begun work on creating a library of callable functions for sequence processing that use classical string processing algorithms (e.g. not using dynamic programming). Among the functions that we have implemented are suffix tree creation and retrieval, and a variety of palindrome and tandem repeat finders. The latter work with an arbitrary number of errors (user parameter), inversions, and single insertions and deletions.

In all cases, the preferred FPGA algorithms are significantly different from the serial versions. Besides the use of parallel circuitry (e.g. to perform multiple comparisons at once) it was important to align the geometry of the processing with the data flow. In such a way, it was possible to simultaneously analyze a string in a variety of ways as fast as the elements could be loaded onto the chip. For example, one FPGA configuration finds palindromes, complementary palindromes, and tandem repeats, all up to length 64, all with an arbitrary threshold on errors, and all with single insertions and deletions. Note that this length restriction was not due to running out of chip area, rather it was because of problems with the synthesis tool beyond that size.

The serial reference code took 14 microseconds per character, the FPGA implementation less than 10 nanoseconds per character. Obviously this particular combination is not likely to be interesting, but this result still shows: the utility of FPGA acceleration for sequence processing and that sequence analysis functions can be efficiently mixed and matched. A further application of these methods could be in motif finding using Markov models.

Dynamic programming for sequence alignment

This is the classic application for FPGA acceleration; 1000x speed-up has already been established [55]. Our design (see Figure 5) differs from other work in that it allows complex programmable scoring functions and takes particular advantage of the on-chip memory configuration of new generation FPGAs. Also, unlike many previous FPGA versions, both forward and backward passes are implemented.

Molecule interactions

This case study is described in detail the next section on future work. The key point to mention here is that the preferred methods on an FPGA are completely different from PC or cluster. The solution uses direct correlation, rotation through coordinate axis transformation, and coordinate axis transformation

through indexing.

Optimization through successive approximation
Successive approximation algorithms include hill climbing, Gibbs sampling, simulated annealing, and learning neural nets. They have been applied to many BCB applications, such as in using Bayesian Networks to analyze gene expression data. They are all efficiently implemented on an FPGA following the same framework. Here is one example for hill climbing. The FPGA is partitioned into a number of processing elements (PEs), generally 100's, interconnected by a reduction network and a broadcast network. On every iteration, the PEs each compute the objective function, either at a different point in the space, or in a different direction from the same point. The results are collected with the reduction network, and the action for the next iteration determined (e.g. after seeing which direction has the steepest gradient). This information is broadcast to the PEs which compute their new points in the space and the new objective function. This continues until the stopping criterion has been reached.

Implementation of common structures
Throughout this section we present numerous examples of recurring FPGA structures. The PI has previously developed algorithms and IP for a number of these, including PEs [30, 49], associative constructs [34, 33], switches [32], and control [31]. Some of these designs involve SIMD arrays: these are themselves applicable to BCB as exhibited by the Kestrel project [25] and so are prime candidates for FPGA configuration.

3.1.3 EDA enhancements

We have done work in automating two areas, precision management (optimizing the datapath width) and speed-matching pipeline stages through replication (both mentioned briefly in [56]). The precision management process is particularly novel: it involves analyzing the arithmetic using probability density functions to represent ranges of possible values or to represent uncertainty due to quantization errors. This is a superset of worst-case analysis; it gives much more statistical information and so allows for substantially more efficiency.

3.2 Programmability

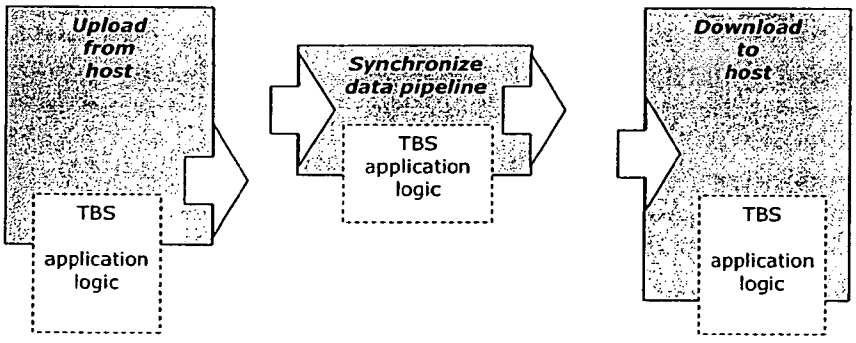


Figure 2: Generic templates.

The purpose of TPS is to bridge the semantic gap between biological computation and hardware design (see Figure 1). The method is to encapsulate knowledge of hardware design for BCB applications from the application programmer. This includes hiding: novel circuits specific to BCB problems, classes of FPGA algorithms specific to same, and the parameterization of those algorithms and circuits. TPS thus masks the hardware environment from the user and so the application is reuasable across new chips, boards and backplanes. At the same time, however, TPS needs to enable the use of the full FPGA capability. To achieve both of these conflicting goals, TPS is based on a template (or object) model the details of which are given in the proposed work section.

A number of template examples are given immediately below; e.g. in Figure 2: the gray denotes what is supplied by the system and white what is to be supplied (TBS) by the user. A template can be a complete solution or a module. Templates can be concatenated to create applications (as in Figure 3). Templates can be fixed or parameterized. Some parameters are simple, such as the specification of data types. However, the power of the TPS comes from templates that allow the user to provide functions and even HLL codes as parameters. Two examples of this (given below) are the objective function in an optimization application and the scoring function in a dynamic programming application.

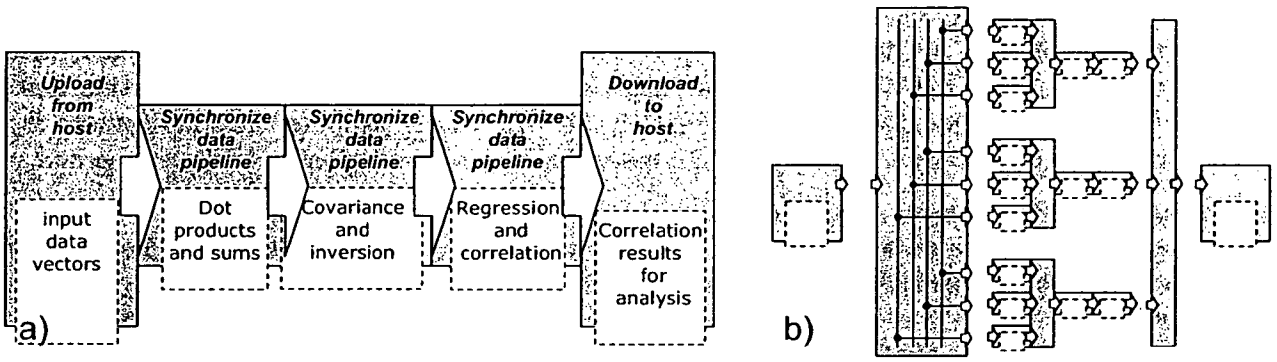


Figure 3: Generic templates have been instantiated, replicated, and concatenated to implement the microarray data regression analysis.

Microarray

Figure 3 shows the microarray data analysis application described in the previous section. Note that in Figure 3a) the user has plugged functions into the system templates and concatenated them. Figure 3b) shows how the modules are replicated and pipelined to use resources and speed-match stages. Also note that the data replication and distribution network is automatically inserted between input and the first application pipeline stage.

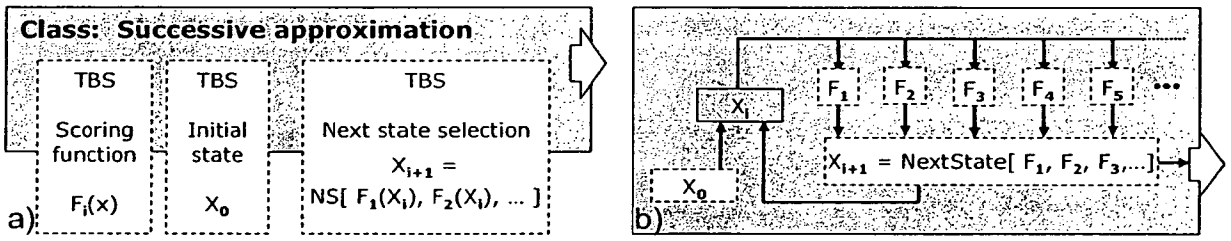


Figure 4: Successive approximation in TPS.

Optimization through successive approximation

Shown in Figure 4 is template solution to the algorithm described in the previous section: a) shows the programmer's view, b) the mapping to the FPGA. The $F_i(x)$ are the scoring functions computed by the PEs, X_0 is the initial state, and so on. The Next State function may have several outputs: the next set of values to be scored, a convergence test, and the answer produced on the final iteration. Note that the programmer only needs to program using familiar concepts and provide appropriate serial code. Also note that this means that the programmer is oblivious to the underlying hardware.

Dynamic Programming

Shown in Figure 5 is a solution to dynamic programming of the type found in Smith-Waterman, Needleman-Wunsch and other algorithms. Figure 5c) shows the naive solution with a direct mapping of the serial algorithm; Figure 5a) shows what the programmer sees (only the cell); and Figure 5b) shows what actually gets instantiated in the FPGA. The user fills in the scoring function while TPS supplies the data sequencing and buffering. On the forward pass, the string streams past logic cells for scoring,

with the backtracking information being queued in high-density on-chip RAM. On the reverse pass, the stream scores the past logic cells for backtracking.

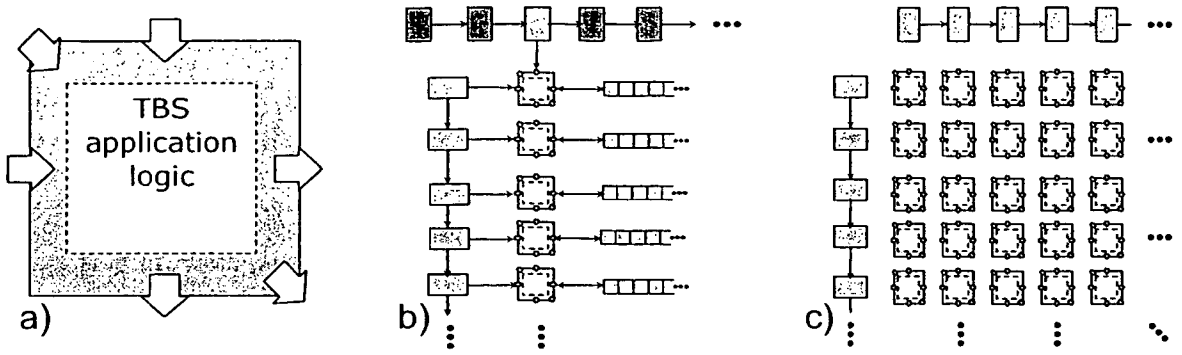


Figure 5: Dynamic programming in TPS.

3.3 Summary of Preliminary Studies

Through the case studies we have shown that FPGA acceleration is promising in more applications BCB than previously realized. We have also gained experience in the use of FPGAs for BCB: this is necessary to design create a flexible software environment that is the core of TPS. Note that the success of TPS depends on there being a finite set of algorithmic templates that encompasses a number of BCB applications. There are at least three reasons why we believe this is true: (i) the results so far are promising, (ii) that this was the case in another domain (mapping computer vision algorithms to array processors [29]), and (iii) the high-level analysis (at the beginning of this section) showing a recurrence of particular kernel functions.

4 Research Design and Methods

4.1 Overview

From our previous work we have derived three results: (i) the match between the computational characteristics of the applications and the capabilities of the technology, (ii) the speed-ups from the preliminary case-studies, and (iii) the apparent efficacy of templates in encapsulating classes of BCB computations. We believe that these results demonstrate that BCB using FPGA acceleration is worth pursuing. Other previous work [55] has shown significant, but very narrow success: in Figure 1 we denote this by the vertical slice from L4 to L0. Clearly two things are needed to advance the technology: to broaden the application base and to broaden the programmer base. Once critical breadth is achieved, these two will feed off one-another and eventually lead to the development of entirely new applications that take advantage the computational capability.

To broaden the application base, we propose work on a domain that has not yet been addressed with FPGA computation: Molecular interactions. This domain is of critical importance while appearing to be good match computationally; it is also different enough from other BCB/FPGA applications that its success will be a significant demonstration of breadth for the technology. To broaden the programmer base we need to demonstrate that TPS is viable. The way to do this is to create a prototype. The third area of work is a continuation of development on applications we have already begun: string processing and microarray data analysis.

4.2 Data Sharing

The results of academic interest will be published in scholarly journals and conference proceedings. All software, circuit designs, and documentation will be made available on the project web site. Because the circuit designs will be in HDL format, this denotes complete sharing.

4.3 Work to be done

4.3.1 Prototyping TPS

Creating the entire TPS is the work of many engineer-years, so a prototype must be constructed to demonstrate its efficacy. The prototype environment must show two things, that TPS is suitable for programmers without hardware knowledge and that TPS yields efficient FPGA configurations.

System architecture
We propose a rich interactive development environment (IDE), comprising a graphic user interface for system composition, template libraries, and compilation tools. Our major criterion is to have a system working at some level, at all stages of development. Our development plan is to put all major system components in place, in rudimentary form, as early as possible. Development will then consist of incremental advances along all fronts of the system specification.

The remainder of this subsection describes each of the major system components. We first describe the component as we envision it in the long run, then the initial implementation, and finally some of the intermediate steps that bridge the gap between the initial and ideal systems. The eventual system will be built to industrial standards of robustness and maintainability. Initial versions, however, will demonstrate basic capabilities using highly generalized prototyping tools. Briefly, the major components of the system are:

1. Graphical user interface (GUI). This will allow programmers to define the application structure in terms of pre-defined templates, and to add their own procedural code into pre-defined templates. It will also connect circuit generation tools back to the graphical representation for reporting application errors and resource utilization in the user's input format.

2. Template and utility libraries. Templates will connect the user's application logic to data streams, to synchronization primitives, to technology dependencies, and to the host interfaces. Utility libraries will provide callable interfaces to low-level functions. Language processing tools will connect the user's procedural code with the template framework and with the accelerator's execution environment. In the long term, these tools will also perform design optimizations. These will include automatic replication of application code up to the capacity of the FPGA, arithmetic precision management, and other FPGA-specific code analysis.

3. Circuit generation interfaces. These will present the completed application to commodity circuit generation tools. They will also extract information from those tools, especially timing and hardware usage, for guiding resource allocation and balancing.

4. Debugging tools. These tools will allow the programmer to simulate the completed application using the same visual and textual conventions that defined the application. Debug aids will be available in the template frameworks, both to support simulation and to extract information from an actual FPGA application.

Graphical User Interface
The eventual GUI will be the programmer's only interface to the proposed system. At the highest level, the GUI will be engineered for extensibility, i.e. to support a wide variety of extensions with minimal change to the GUI core. Wherever possible, the GUI will be built from existing tool suites, possibly MatLab's SimuLink or Xilinx's SysGen. This strategy is meant to reach very high levels of UI quality, versatility, and stability with a minimum of redundant development.

The very earliest proof of principle tools, however, may use text-driven interfaces. Once each sub-

system's central behavior is understood, it will be rephrased as a subsystem within the GUI framework. It is expected that the GUI itself will develop over time, starting with simple presentation of the most important development features.

Template and Utility Libraries

The template and utility libraries bracket the application's unique features. Templates will contain the application logic, managing its IO, connectivity between logic elements, and performing synchronization. As in modern programming environments, locus of control will reside in the framework outside of the application. The framework will invoke application features when data availability and synchronization permit. Utility libraries, on the other hand, are functions meant to be called by the application logic to provide complex or highly optimized primitives.

The highest level of template will mask the hardware details of the FPGA accelerator from the application, including the host interface and the specifics of the chip resources. Top-level templates will provide standardized interfaces to the system environment, parameterized in application-specific data types.

Templates wrap system-specific logic around application-specific code elements. Those elements may be procedural definitions of the application's function, or may be instances of other templates. Recursive use of templates is an essential feature of the system: it provides effectively infinite flexibility of application structure using a relatively small number of template building blocks. Note that recursion, in this case, refers to the rules by which template-based systems may be composed. Any real design will use some specific number of template instances, layered to some finite depth.

Early versions of the system will not support user-defined templates. In the long run, though, user-defined templates will be implemented. This will increase the system's programmability by a significant step and will reduce the user's dependence on the system implementor.

Strategic templates will provide high-level logical frameworks for building applications, within the framework set by the system template. Many applications fall broad categories, including:

- Successive approximation and other families of algorithms (as described above),
- Data streaming, with or without double buffering, in which the host downloads relatively large amounts of data and uploads results continuously while the application runs, and
- Interactive, e.g. transferring low-volume requests and responses between the host and the accelerator, possibly after transferring a large amount of setup data.

Low-level templates allow system composition from elements such as: pipelines, built up from elements that process data consecutively, in a bucket brigade fashion; parallel computation, multiple elements to execute concurrently; vector operations, including custom reductions or dot products; specialized combinatorial networks; and so on.

Utility libraries will include optimized function blocks without user-definable operations. One such block might accept a small matrix as input and produce its inverse as output, another might perform matrix multiplication, another might perform some kind of vector operation. Such blocks might have parameters representing data types or control constants, but few user options.

Language processing tools.

The principal way for programmers to specify detailed application will be through use of a procedural programming language. Scripts or function definitions will customize the behavior of templates. Templates will provide a flexible interface at each position where custom behavior is accepted, including the inputs to that template slot and outputs from it. Inputs and outputs will commonly allow user-defined data types, allowing very flexible designs.

In the long run, the procedural language should represent algorithms at a high level, comparable to Java or Forge. These languages support rich semantics, while also allowing many kinds of automated

analysis. Eventually, the language processor should be able to interact with the template framework in terms of the system's technology parameters. This will allow partial or complete automation of many design tasks, including: maximizing parallelism, subject to available hardware resources; rate-balancing pipelined operations that run at different operating frequencies; and selection of arithmetic value widths, subject to input constraints and precision requirements. Utility libraries will also provide functions callable from the procedural application code.

It is not clear that any existing hardware design language will meet this project's technical requirements. Eventually, this project will expand to include compiler and possibly language development, perhaps based on the SUIF compiler development tools. We will, however, review the suitability and feature set of languages such as Forge, Handel-C, and System-C.

Initially, however, this project will use some existing design language. That may be a VHDL subset, augmented with custom libraries and macros. Early experience with an existing language will help define the requirements for a more self-contained system, and will add real programming capability to early prototype systems.

Visual program-composition tools exist in which *all* development uses visual tools. Every arithmetic operation, every constant, and every variable is represented by some visual block. We feel that, at this level of detail, typical programmers would be frustrated by the low density of information on screen and by the relative complexity of making simple changes. On the other hand, visual templates make it easy to select and tailor high-level features without presenting unwelcome detail about the hardware-specific implementation. The combination, visual tools for high-level assemblies and procedural code for low-level functions, may make optimal use of the programmer's attention and skills.

Personnel and Schedule

This work will be conducted by Tom Van Court, a graduate student and software engineer with twenty years experience in designing and building large software systems, together with Yongfeng Gu and the PI, and in consultation with engineers to be determined at Xilinx, Inc. We project the following three month milestones over two years: i) bring up hardware development system, initial system template for I/O, ii) working skeleton program for accessing system template on FPGA, initial template library (single application, pipelines, inclusion of user code), iii) non-GUI tools flow established, initial language processing, iv) synthesis tool flow (first application built using complete non-GUI tool flow), v) create first templates that replicate logic parametrically, initial GUI demonstrated, vi) full GUI tool flow, extended template library, vii) performance usability testing.

4.3.2 Case Study: Protein-Ligand Docking

In grid-based protein-ligand docking [37], the algorithm divides the space containing a substrate molecule into a three-dimensional grid of space elements (called voxels). Each grid cell is tagged as being inside or outside the substrate molecule. Grid cells may also be tagged with other data of interest: whether or not the cell is at the surface of the molecule, hydrophobicity of the surface element, nearby electrical charge, and so on. The ligand molecule is also broken down into tagged voxels, using the same grid pitch but possibly a different number of grid cells. The ligand grid is then tested against the substrate grid, in many relative translations and rotations, for favorable alignment of molecule surfaces and for unfavorable collisions of molecule interiors.

On first examination, grid-based docking [16], seems an unlikely candidate for FPGA acceleration. That technique represents voxels as complex numbers. Molecule surfaces are positive values, exteriors are zero, and interiors complex. In 3D correlation, surface contacts yield positive scores and interior collisions yield negative scores - high surface contact without overlap scores best. The correlation is repeated at large numbers of (x, y, z) offsets and 3D angular displacements. The final result of the calculation is some set of offsets and angles at which the highest correlation values are found.

To reduce the polynomial order of the computation, each grid of complex numbers is subjected to a 3D Fourier transform [37]. Transformed grids are multiplied, and the result is back-transformed. This is repeated for many rotations of the ligand grid with respect to the substrate. The transform step sets the polynomial order of the calculation, and is the conspicuous candidate for acceleration. Unfortunately, it would take a very elaborate hardware design to perform the 3D complex floating point transform, and would probably yield modest speedup, if any.

The basic features of the problem seem amenable to acceleration, however. The original data values are signed integers of just a few bits. The basic computation, multiply and accumulate, is simple and can be repeated many hundreds of times within an FPGA. If modest grid sizes, 32^3 to 64^3, are used for an initial scan of configuration space, the whole model will fit on chip or in dedicated RAMs. That gives fast access to operands, and multiple RAMs increase the total data access rate even more. Also, *ligand rotation can be combined with voxel grid addressing* eliminating the separate, serial step of rotating its voxel grid. This also eliminates the need to reload the ligand grid for each orientation. Correlation is well-studied in the signal processing literature [21]. That means that many efficient computation structures exist for performing the correlation directly. Finally, the FPGA can parallelize the search for the best few (x, y, z) offsets at a given rotation. That eliminates yet another serial step needed by the transform-based approach, and may even eliminate output and storage of the 3D correlation grid.

The transform approach seems preferable to the direct approach because of its $O(N^3 \log(N))$ versus $O(N^6)$ complexity (where N is the number of steps along each axis). However, for many N, a direct correlation tuned to an FPGA's strengths should give answers substantially faster than the transform approach: e.g. it is 128-bit floating point on the one hand versus thousands of small, parallel, operations on the other.

Personnel and Schedule
This work will be conducted by Yongfeng Gu, a graduate student and former researcher at Intel Research Labs, together with the PI and with consultation from Sandor Vajda. We project the following 3 month milestones in the first year: i) demonstrate and tune axis rotation addressing, ii) correlation and collection of maxima, iii) reimplement using templates – initial demonstration, iv) tuned TPS-based demonstration and performance testing. After that, many different directions are possible in adding complexity to the model.

4.3.3 Other applications

In the preliminary studies section we described work on several applications including microarray data analysis, string processing, dynamic programming, and optimization through successive approximation. Continuing this work will serve the dual purposes of expanding the application base in popular directions and increasing our experience in a way vital to the construction of TPS.

Personnel and Schedule
This work will be conducted by a graduate student to be recruited and by undergraduate students together with the PI. We also anticipate several senior group projects being derived from this work. Many of the basic functions are implementable in a few months by the skill level available. During the period of the project, it should be straightforward to complete and test: the linear discriminator and at least one other microarray data analysis application; several variations of the dynamic programming automaton; a few variations of successive approximation; and several additional string functions.

Environment for Creating and Running BCB Applications on FPGA-Based PC Accelerator Cards

1 Description of Technology

1.1 Overview

The Human Genome Project and related work have transformed Biology from a purely laboratory-based science to an information science as well [5]. Moreover, this successful execution of 'Big Science' has enabled "the key to scientific progress[, which] is to unlock the brain-power of the individual researcher. [6]" This empowerment manifests itself primarily in the capability of scientists, from their desktops, to access huge databases and to there conduct rudimentary processing of that data.

This empowerment has not extended nearly as much to computation: although much can be done with a desk-top PC, there are also severe limitations (which are discussed below). The cost of these "beyond-PC" computations in terms of system acquisition, learning curve, maintenance, and convenience has limited "Big Computation" to large research projects. The most notable of these is perhaps the aforementioned assembly of the human genome by large farms of thousands of workstations.

We propose the logical next step: to bring to the individual researcher a capability in computation analogous to the existing capability in data access in terms of speed, cost, convenience, and flexibility. The mechanism is to augment the PC with a low-cost FPGA-based computational coprocessor; that is, a plug-in board similar in cost and ease of installation to a graphics card. The key, however, is not the hardware itself; rather it is to supply with this "accelerator" a computational environment that enables its efficient use. This environment will support a wide range of user sophistication by including:

1. Canned applications (such as BLAST) for users with no computer background,
2. Application/Function libraries for users with scripting or spread-sheet programming capability,
3. A High-Level Language for programmers, and
4. A Hardware Description Language interface for FPGA/circuit designers.

The basic deliverable is a factor of 100 to 1000 speed-up over a PC with a materials and manufacturing cost of between $2000 and $5000.

The advantages of speed, cost, and convenience are obvious; flexibility less so. This system will not only allow current large-scale applications to be run at the desk-top, it will break the chicken-and-egg phenomenon in algorithm development whereby most application developers avoid computationally complex algorithms because the appropriate hardware is out of reach. Putting it *within* reach will allow the community of developers to apply algorithms with complexity greater than $O(N^2)$, the nominal limit for everyday PC computations with the expected data sets.

A further advantage of flexibility is derived from the fact that bioinformatics and computational biology (BCB) do not occur in a vacuum: you generally can't just throw computation at a problem. Rather, the norm is develop and run computations in close collaboration with biologists. A typical scenario involves a feedback loop where biologists formulate problems and interpret results while computer specialists create and tune the applications as directed by the underlying biology. Essential to the functioning of this development loop is agile computing.

Although we have thus far emphasized desktop PC computing, this system is equally applicable to clusters of PCs. In that case, the goal is to use the accelerator to address true Grand Challenge computations, e.g. such as in molecular dynamics. The potential impact therefore is (at least) two-fold:

- To empower the individual researcher by reducing "heroic" computations that would normally take weeks to minutes. This has two consequences, cost reduction (supercomputer on the desktop) and the potential for developing new and qualitatively different algorithms.
- To create knowledge through brute computational capability.

1.2 The Market

1.2.1 What is BCB? Which part of it are we addressing?

To get an idea of the massive computational problems in BCB we look at one aspect, the generation of ever more massive amounts of biological data. According to a briefing by Sun Microsystems, "[B]CB ... is called upon to clean up the mess created by the recent collision of three major paradigms: robotics, miniaturization, and parallelization. [7]" Again quoting from the Sun briefing: "The result is high density experimentation resulting in the generation of so much data that humans are no longer able to comprehend [it] ... and take appropriate action without the aid of computers. [7]"

More generally, Bioinformatics and Computational Biology (BCB) is simply the use of computers in the life sciences and engineering. According to Frost & Sullivan, the IT needs of biotech companies are the following: databases, analytical applications, middleware that links information from various separate databases, and systems that store and distribute information [1]. The aspect that we address is analytic applications. Some of the broad domains within "analytic applications" are: genomics, functional genomics, proteomics, pharmacogenomics, biosimulation, combinatorial chemistry, and drug design. Within each of these domains are many applications; e.g. a sample from genomics includes genome assembly, gene identification searches, motif searches, genome characterization, and comparative genomics. Applications often share computational characteristics with others within and across domains, but also often vary greatly even within a domain.

For space reasons we will cut our description of BCB short at this point. As pertains to this proposal, suffice it to say that so far we have found no analytic application for which the FPGA acceleration by the factors quoted above is not applicable. Below we will briefly describe our progress in three domains: string processing (genomics), microarray analysis (functional genomics), and protein docking (drug design).

1.2.2 Market Size and Growth

Although we have not conducted a formal market survey, there are many indications of market size and growth. Looking first at overall BCB, one indication is the expenditure of many large corporations in this area: Intel, IBM, Microsoft, Oracle, Sun, and HP all claim to be spending hundreds of millions per year on BCB product development. Another indication is the hundreds of smaller companies in the area [1]. Still another is derived from a broad analysis of the health care sector: it continues to grow faster than the economy as a whole and BCB is taking a increased fraction of that sector. Just one of many possible examples of this is that drug companies are rapidly increasing the use of computational screening early in the drug development process to improve both time-to-market and the chance of success. With concerns about homeland security, this trend should continue.

As pertains specifically to analytic applications, one indication of growth is that some new biomedical technologies that require substantial computing support are experiencing phenomenal growth: one example is microarray assays whose use is growing at 70% per year. Another area with potential for explosive growth is *in silico* simulation.

1.2.3 What is BCB going to be? How will it evolve?

With the completion of the human genome project, it is generally agreed that emphasis will move from description to function, structure, dynamics, and systems. At the highest level, this represents an extension of computation from one dimension (strings/sequences) to three (molecules in space) or four (space plus time). Interactions increase the complexity still further. [1]

These new problems and new technologies generally require different classes of algorithms than those central to genomics. That field is dominated by string processing, especially classical string algorithms, suffix tree algorithms, and dynamic programming. Classes of algorithm that appears to be central to the new

[1] Of course these complex computations are not new, only their relative emphasis.

emphasis are searching high-dimensional parameter spaces, linear algebra, and molecular dynamics. Again for brevity we curtail the discussion here, stating only that these applications require, with the envisioned data sets, a sizeable leap in computational capability.

1.3 Competing/Supporting Technologies

A summary of currently used technologies follows.

1.3.1 Non-FPGA Based

1. PCs — PCs are cheap, distributed, and flexible. There is much free and commercial software and PCs have a familiar programming environment. It is likely that the vast majority of BCB application executions occur on PCs. The drawback is computational power: only algorithms up to O(N**2) complexity for common data sets are generally run. This means that PC applications make liberal use of assumptions and heuristics to limit computational complexity. Also, the complex higher order computations described above are completely out of reach for PCs.

2. PC clusters — PC clusters are popular and becoming more so with many thousands in active use. The performance is up to P-times better than a PC for P processors. For small clusters (fewer than 16 nodes), the material cost can be slightly less than P-times that of a PC. Larger clusters, however, e.g. those with more than 64 nodes, have special housing, network, power, and cooling requirements. In all cases, there is a significant system management cost with a full-time-equivalent system manager being the norm even for small clusters. There is much less available software than for PCs and skilled programming is required to get good performance.

3. Large centralized facilities — These come in many varieties and include the generic supercomputers and MPPs at the national centers and the huge workstation farms at some companies. As with PC clusters, performance is up to P-times better for P processors. However, it is more likely that this performance scaling will extend to larger problems. Also, the programming model may be simpler. However, the cost is very high, both in acquisition and system maintenance. Researchers often apply for supercomputer time from public facilities: however, this rarely appears to yield a very large benefit. Applications require initial scalability studies, programs must be submitted batch with long turnaround time, and rarely is the user permitted more than a very small fraction of the resource.

4. Other ASIC solutions — Periodically researchers have created Application Specific Integrated Circuits (ASICs) capable of tremendous performance for certain applications. It is almost always the case, however, that they are obsolete almost as soon as they are built.

1.3.2 FPGA Based

The only FPGA-based products for BCB computation are turn-key systems sold by TimeLogic [8] and Paracel. These consist of clusters with dedicated accelerators whose hardware is similar to what we are proposing here. For the set of applications that are supported, very high speed-ups are achieved. The drawbacks are cost and flexibility. Minimum configurations start at $70,000 and typical systems cost $500,000 and up. Perhaps even more importantly, these systems can *only* run those applications supplied by the vendor. Currently there are about ten of these.

Another potential competing technology is general purpose FPGA hardware and supporting software. General purpose FPGA boards are sold by many vendors including Avnet, Nallatech, and Annapolis Microsystems. These are generally used for evaluation and prototyping before system integration. Such boards could provide the hardware substrate for this product. When configured appropriately, tremendous speed-ups can be obtained, as with the turn-key systems just described. The problem (addressed here) is that configuration for high-performance applications requires skilled application analysis, algorithm development, and circuit design. Much preferred would be that an application programmer or end user could obtain similar results. Much work has been in done in developing supporting Electronic Design Automation (EDA)

software toward this goal. The most successful products generally fall into one of the following categories:

- **IP.** Because circuits are created in software before they are cast into an ASIC or configured into an FPGA, it is trivial to reuse the design. Designs, called Intellectual Property (IP) blocks can therefore be sold, either individually, or bundled into libraries. Most often, IP blocks are nitty-gritty system components such as bus, I/O, and memory interfaces, or embedded microcontrollers.
- **Hardware Design Language (HDL) code from High-level language (HLL) code.** Examples are Handel-C and Forge. These have proved extremely useful in design management, coupling of simulation and design, and hardware/software codesign. They have not, however, achieved the goal of automatic compilation of generic HLL code to efficient circuits.
- **Domain Specific GUIs.** Xilinx has a product called System Generator for DSP. It allows the user to design circuits via data flow. That is, the designer can use a GUI to specify how signals flow through various components such as converters, filters, and transformers. These components are commonly IP blocks. The drawback of this system is that it only currently works for DSP. Creating such a system for BCB is one of the goals of the current project.
- **Domain Specific HDL code from HLL code.** This approach has met with success with the application of the end-product is constrained. An example is the SA-C language and compiler developed by the Cameron Project for computer vision processing hardware. Constraining the domain allows the HLL to be restricted, making the problem of converting HLL to HDL tractable. The drawback of this system is that in only currently works for computer vision. New language constraints would need to be developed for BCB. Creating such a language and compiler is one of the goals of the current project.

1.3.3 Summary of Other Technologies

Of these existing technologies, only large-scale MPPs/Supercomputers and dedicated hardware provide high performance. MPPs/Supercomputers are extremely expensive, inconvenient to use, and not easy to program. Dedicated hardware is expensive and not programmable except by the vendor. The ideal solution, of course, combines the best of all of these: the low cost, convenience, and programmability of the PC with the high performance of the MPP/Supercomputer or the dedicated hardware. We believe that this can be achieved by creating systems for BCB that are analogous to SA-C and System Generator for DSP.

1.4 BCB Computation Using FPGAs

In this subsection we justify the use of FPGAs for BCB computation. We begin by introducing FPGAs, sketch the characteristics of BCB computations, show why the latter maps well to the former, and give performance examples.

1.4.1 What is an FPGA?

Field programmable gate arrays (FPGAs) are integrated circuits whose (apparent) circuitry can be determined, or *programmed*, in the field. This is in contrast to ASICs whose circuitry is fixed at fabrication time. The tradeoff for this flexibility is that FPGAs are less dense and fast than ASICs; however, for many applications the reduced engineering and manufacturing cost and drastically improved time-to-market more than make up for the drawbacks. Beyond the configurable substrate, current generation, high-end FPGAs also contain optimized modules, including entire microprocessors. To summarize, high-end FPGAs have the following characteristics.

- Programmable in milliseconds by uploading the desired configuration. This also means that the FPGA can be *reprogrammed* for other applications just as quickly.
- 4 million+ configurable gate-equivalents
- Millions of communication paths, both local and global
- Circuits are generally designed using hardware description languages (VHDL, Verilog); much available Electronic Design Automation (EDA) support

- Design modules often available as "intellectual property" (IP) blocks
- Hardwired on-chip gigabit interfaces (Infiniband, Gigabit Ethernet, etc.)
- Hardwired on-chip microprocessors, busses, and memory modules
- Hundreds of hardwired 18-bit multipliers

For the right applications, speed-up factors of from several hundred to a thousand or more are common. Currently, most FPGAs are used in Digital Signal Processing (DSP) applications.

Besides the engineering reasons for using FPGAs, there is a business reason as well. FPGAs are already a major industry and as ASIC fabrication becomes ever more expensive, their market share is likely to continue growing. We therefore satisfy a fundamental requirement of computer-based products: a method of "riding the technology curve" as stated in Moore's Law. That is, FPGAs will continue developing at rates comparable to microprocessors. Therefore, as new generations of FPGAs emerge, existing *hardware* designs can be ported to them in much the same way that software can be loaded onto a new generation computer.

1.4.2 Characteristics of BCB Computations

We have analyzed a number of applications, but for brevity, describe only two: genomics and functional genomics through microarray analysis. We then summarize the application characteristics.

The fundamental assumption of BCB is that biologically meaningful results can be derived by abstracting physical phenomena into data structures and algorithms. In genomics, the physical entities are the amino acid sequences of proteins and the nucleic acid sequences of DNA and RNA. The data structures are character strings; the algorithms process those strings. In this way, string analysis problems correlate with biological problems. Here are some examples with first the string question, then the corresponding biology question:

1. **STR:** Reconstruct long strings from overlapping fragments
   BIO: Sequence DNA
2. **STR:** Compare strings, exact or inexact
   BIO: Find whether two genes related?
3. **STR:** All-to-all string comparisons
   BIO: For two organisms, find which genes they share
4. **STR:** Compare a string with strings in a database
   BIO: Find what other organisms have this gene
5. **STR:** Does a particular pattern occur in a string?
   BIO: Look for recurring protein structures
6. **STR:** Characterize the differences between strings
   BIO: Find how has a gene has changed over evolutionary time

The algorithms fall into a relatively small number of classes: exact and inexact string matching using classical techniques, partial string matching using dynamic programming, string structure analysis using Markov models, and string comparisons and analysis using suffix tree techniques. Data set characteristics are as follows: they have a small alphabet, from to 20 letters; there are 10s to billions of characters per string; there are from 1s to millions of strings; the data are noisy; and comparisons often require evaluation with a match table with size 20 x 20 x score. The computational complexity of some sample algorithms is as follows. For n characters and k strings: simple matching using suffix trees is O(kn), partial matching using dynamic programming is $O(n^k)$, while assembly can be O(exponential).

Moving on to functional genomics, one popular technique involves using microarrays to measure simultaneously the expression products of thousands of genes in a tissue sample. The individual data points, however, are very unreliable. A microarray study consists typically of at most 100s of microarrays, e.g. from patients or experiments. Here the fundamental assumption is that biologically meaningful results can be derived from relating expression levels to sample phenotype. Here we again give sample questions, first assuming m vectors of size n and then the biological equivalent assuming m genes and n tissue samples:

1. **ALG:** Which size-k sets of m vectors correlate best with the outcome?
   BIO: Is there a group of k genes of the m we are observing that can serve to distinguish the outcomes of patients with disease *ijk* who are otherwise indistinguishable?
2. **ALG:** Cluster all m vectors to create a dendrogram
   BIO: Which genes are expressed in unison?
3. **ALG:** Assuming that groups of the n samples are taken at various times during a process, how is the expression sequence related to the outcome?
   BIO: What distinguishes the molecular pathways for signaling of two of the substrates of the insulin receptor?
4. **ALG:** Search a database for best matches
   BIO: Which of all known genes have regulatory mechanisms that appear to be similar to those regulated by the *lmnop* transcription factor?

One observation is that the correspondence of computation and biology is not as close in microarray analysis as it is in sequence analysis. It is perhaps not surprising that a wide variety of techniques are used. First, what does *not* work is standard biostatistics. Data are invariably underconstrained with many more variables than samples. However, these huge amounts of data appear to be able to tell us useful things!

Some classes of algorithms used are as follows; there is not a fixed correlation of algorithm to question: clustering, dendrograms, applications of standard statistics after preprocessing, self-organizing maps, Bayesian nets, and AI inspired machine learning. However, the data sets are remarkably uniform: expressions require 2 to 4 bits, outcomes = 1 bit; there are 10s of thousands of genes and up to 100's of samples; the data are very noisy with some data are missing altogether. Again there is a wide variety of computational complexity from $O(n)$ to O(exponential).

Some characteristics we can derive from these and other BCB applications are as follows.

1. Low resolution data — Data elements require only few bits. This is true of sequence data, microarray data, and even molecular dynamics if the problem is cast the right way.
2. Large but manageable data sets. There are thousands of genes, thousands of samples (at a time). When the data sets are very large, the data are accessed sequentially such as when streaming through a database. Other computations are inherently I/O bound.
3. High-dimensional parameter set that must be searched or enumerated to identify a solution. Many preferred algorithms have very high complexity.
4. Simple performance criteria (score functions) that is evaluated for each candidate parameter vector. While the algorithms may have very high computational complexity, the individual computations tend to be quite simple.
5. Decomposable search strategy. Many algorithms are easily partitioned into multiple independent problems.

Computational complexity is only part of the challenge of BCB Computing: there are a broad range of rapidly evolving computationally intensive domains, a growing number of analysts and researchers in academia and industry, and rapidly developing biotechnology — new problems, new approaches, new algorithms, new instrumentation. All this again points to flexibility as well as power.

1.4.3 Using FPGAs for BCB: Speed-Up Case Studies

The utility of FPGAs for some BCB computations has been proven. Existing products such as from TimeLogic have achieved speed-ups in the 100s for their application sets [8]. For one dynamic programming application, these have been surpassed by a university project seeking an inexpensive alternative [10]. In our own work, we have shown a factor of 1000 speed-up on a generic functional genomics application [4, 9]. We have also examined a number of other applications (see the IP section) and anticipate similar speed-ups there as well.

1.4.4 Using FPGAs for BCB: General

Here we give a high-level argument why this speed-up trend is likely to continue through a number of BCB applications we have not yet examined. In other words, where does the factor of 1000 come from? Why is there a good match? In architecture good performance is dictated by fast cycle times, but at least as important are parallelism and locality. In other words, how many processing units can we simultaneously bring to bear and how long does it take to get the data them. It is often the case that these goals are at odds with one another; in the mapping of BCB to FPGAs they can often be achieved together. Some of the reasons are as follows.

Data elements have a small number of bits — This is true in sequences, microarrays, and even molecular dynamics when cast appropriately. Since only a few bits need to be processed per datum, we can configure the gates of the FPGA into many thousands of Arithmetic Logic Units (ALUs). This is in contrast to PCs where there are a small, fixed, number of large ALUs. The latest FPGAs also have optimized dedicated circuitry to support hundreds of parallel multiplications.

Data set sizes and management — For many applications, the data sets are on the order of a few MB. This fits comfortably on-chip allowing for entirely local computation and avoiding time-consuming off-chip transfers. For the computationally intensive applications that are our target, there is massive reuse with each element generally being used at least $O(N^2)$ times. Many large data sets, such as gene databases, are accessed sequentially: in that case the data can be streamed onto (and through) the FPGA at Gb rates through the dedicated I/O structures.

Simple processing kernels — Many computations are repetitive with relatively simple processing kernels being repeated very large numbers of times. Examples of this appear in various applications especially those in finding Hidden Markov Models, Bayesian Nets, Dynamic Programming, and various linear algebra computations. For these computations, FPGAs can be configured into a number of processing elements (PEs) specially tuned to the task. PEs therefore compute very efficiently while only requiring small amounts of chip area. The latter can afford a high degree of replication.

Dataflow — Most of the applications listed above are dominated by regular communication: on any iteration, data only need to be passed to adjacent PEs. These short paths minimize communication time.

Associative Computation — Many of the remaining communications are associative operators: broadcast, match, reduction, and leader election. These occur in the following ubiquitous scenario. Parallel computations need to be collected to see which yielded the current best solution — a leader needs to be elected. As the computation continues into the next phase, the leader broadcasts a new set of parameters to the rest of the PEs. Alternatively, a phase ends with data from all the PEs begin combined (e.g. summed) to form a solution or to determine status (e.g. done/not done). In all of these cases, FPGAs can be configured to execute the associative operator using the long pathways on the chip. The result is that rather than being a bottleneck, these associative operators afford perhaps the greatest speed-up of all.

Other Fine-Grained Parallel Hardware Models — So far we have described non-programmable components. FPGAs can also be configured into stored-program parallel processors. Several standard models are possible, most of which have been well-studied. Three are: (i) SIMD arrays, i.e. a collection of simple PEs that execute programs lock-step on their own data and usually connected via a mesh network; (ii) systolic arrays, similar to SIMD arrays but often having even simpler PEs and used for algorithms with regular dataflow; and (iii) fine-grained multicomputers, i.e. collections of bare-bones microprocessors. Each of these is efficient for a particular classes of algorithms.

Note that these factors are generally not "in the toolbox" of the typically programmer. This leads to two issues that are fundamental to the proposed product. The first is that software constructs or functions must be provided to enable their use. The second is more complicated: applications *must use these constructs to obtain maximal performance.* That is, FPGAs present a computational model that is fundamentally different than that presented by a PC or other standard computer. What this means is that, for any application, it is likely that the algorithm for its optimal execution will be different for FPGA than it is for PC. And in particular, it is likely that the FPGA algorithm will involve these constructs while the PC algorithm will

not. Consequences of this are discussed below.

1.5 Product Overview

The product has both hardware and software components. In fact because the hardware is programmable it is sometimes hard to distinguish between the two.

1. **General:** Hardware/software compute engine installable and upgradeable in a PC by lay person. The idea is for the hardware to be completely transparent with the software providing usability at the level appropriate to the user capability. Although more user effort and experience will yield broader product utilization, there should be plenty for everybody who does BCB computation. An analogy is a graphics card: although it provides an essential function, it is transparent to PC users with only graphics programmers (e.g. video game builders) accessing it directly.
2. **Hardware:** The hardware is a high-end FPGA chip on board with a standard high-speed (PCI) interface. An existing third party board (e.g. from Avnet or Nallatech) may be sufficient. However, development may determine that a special memory or I/O interface is necessary and a board would then have to be designed.
3. Software interface: PCI driver and standard GUI.
4. Provide a hierarchy of solutions depending on user sophistication:
   (a) Complete applications for users with almost no computer background. This is the level of support provided by TimeLogic and most of the same applications will be provided here.
   (b) To serve users who need more BCB applications than the ten or so provided whole, an application libraries and design flow editor will be provided. The idea of this component is to provide a product for BCB analogous to Xilinx's System Generator for DSP. Among the functions to be included at this level are modules for data access, string processing, microarray analysis, linear algebra, and associative operations. At this level, users should be able to mix-and-match components to create a large number of custom applications.
   (c) To allow users to create their own modules, a domain specific high-level language that compiles to the hardware. The idea is to create for BCB what SA-C provides for computer vision hardware.
   (d) Hardware description language interface. For users with moderate hardware sophistication, e.g. at least an undergraduate degree in computer/electrical engineering with an interest in creating highly tuned components, or IP modules for distribution or sale.

The potential benefits are as follows.

1. High performance. Application-wide performance for at least some applications of a factor of 1000 has been proven. Consistent factor of 100 performance benefit is very likely.
2. Low cost. Materials cost is in the neighborhood of $2K.PCBengineeringis0$ if we go with a board vendor and low if we do not. Just as important, no system management cost and relatively low programmer cost.
3. Flexibility. Unlike other solutions currently based on FPGAs and ASICs this is easily extendable to new applications and new implementations.
4. Distributed. Unlike high-cost centralized multicomputers, this solution is low enough cost to go on every desktop. This is one of the key ideas: empowering the individual researcher and freeing them from the requirement of sharing a centralized facility.
5. Local control. No need to apply for compute time, develop scalability studies, submit batch jobs, wait for turn-around, etc.
6. Development of global collaborative environment. As with PCs, many users will make their own software publically available. It will also be in the interests of BCB software houses to develop versions that support this environment.

To summarize: the goal is to provide what current FPGA-based turn-key products provide, but at lower cost and with the capability to extend the utility to many, if not most, BCB computations.

2 IP Developed and in Progress, Current Status of Invention

2.1 Overview

The core IP is a software system that enables BCB researchers to create, with a modicum of effort, FPGA configurations (circuit designs) that yield speed-ups of a factor of 1000+ for a wide variety of BCB applications. Two transformations are made.

1. **Circuit specification to circuit implementation.** This transformation is normally the realm of the circuit designer. Since good circuit designers are more rare and expensive than even good programmers, the value of a system with this capability is clear.
2. **Problem specification to circuit specification.** This transformation involves creating algorithms. Note that this is much more than simply "mapping" an existing algorithm to hardware: invariably, this transformation means *changing* the algorithm. Why is this? Why can't an existing program or algorithm specification simply be transformed to a circuit specification? *Because the target hardware, i.e. the FPGA, is fundamentally different from a standard computer* (e.g. a PC, workstation, or cluster). And, for many applications, **fundamentally different hardware requires a fundamentally different algorithm to obtain maximal (or even tolerable) performance.**

So two keys are: transforming the application into the right *FPGA* algorithm to create a circuit specification and transforming the circuit specification into an efficient circuit.

There is also a third key: automating the efficient allocation of resources in ways specific to the use of FPGAs for computation. This is in addition to the optimizations already done during synthesis and place-and-route. One example is precision management, the "right-sizing" of the computational units to minimize their chip area and maximize the potential for parallelism. Another is speed-matching among phases of the computation: replicating units appropriately to avoid bottlenecks in data flow.

The IP therefore has several aspects.

- The functions and modules that bridge the gap between the programmer and the FPGA. To the programmer they look like HLL function calls or GUI computational blocks; underneath they contain circuit specifications optimized to FPGAs. They are IP in that they require creation of FPGA-specific algorithms in addition to circuit designs.
- The language design. There are two parts to this. The first is a set of templates to construct application specific modules. The second is the constraints and features that allow the use of standard compiler techniques in creating high-quality circuits by non-circuit-designers.
- FPGA applications themselves that contain new FPGA-specific algorithms (i.e., that have not already been created for other systems)
- Key aspects of the compiler, i.e. software components not typically found in compilers (and completely hidden from view) that enable the creation of high-quality circuits by non-circuit-designers. One example is precision management.

Below is a list of IP we are developing, but first a brief explanation of our development strategy. Our first goal has been proof-of-concept: i.e., the demonstration that FPGAs can get sufficient speed-up for a sufficient number of BCB applications to make such a product plausible. Our second goal has been to investigate the components/modules that should be part of the function libraries. The third goal is to design the language and compiler. The IP listed below falls mostly into the first two categories.

1. Bioinformatics language and compiler
2. Datapath width sizing
3. Parallel access of vector combinations
4. Protein docking computation
5. Detection of tandem repeats and palindromes
6. Suffix-tree-based algorithms

7. Implementation of certain optimization methods with applications in microarray analysis
8. Implementation of K-combination microarray analysis through linear regression
9. Microarray analysis techniques using K-combinations and heuristics
10. Environment and structures for FPGA implementation of a variety of microarray analysis techniques
11. Single chip SIMD architecture

2.2 Bioinformatics Language and Compiler

Context/Problem
Bioinformatics and Computational Biology are obviously important applications. It is apparent from our work (as well as reported results from Time-Logic etc.) that speed-ups of 100x to 1000x plus are possible when using FPGAs to accelerate computations. The problem is to supply design capability transparently through a programming environment/language.

State of the art
Specific limited applications are available at high cost. Existing general purpose languages that compile to circuits either (i) require circuit design expertise, (ii) yield poor performance, or (iii) are for particular non-BCB applications.

Brief Description/Method
The most basic problem in hardware acceleration of BCB algorithms may be the semantic gap between biological reasoning and hardware design. Effective hardware acceleration generally seems to require some rephrasing of the biological problem, using terms that do not come naturally to biological scientists. The basic job of a language designer, then, is to encapsulate knowledge of hardware design, including novel circuits specific to BCB problems and the parameterization of those circuits, in linguistic terms natural to BCB programmers.

Language design will follow contemporary object-oriented philosophy. Major features of the new language (TNL) will include:

- Syntactic similarity to Java,
- Class libraries for application support,
- Modern software design techniques, and
- Support for current best practices in application design.

References: CAAD Lab Notes TVC030701, MCH030612, and TVC030525

Status (Implementation, verification)
TNL is going to be continuously evolving as BCB evolves and so may never be "fully" implemented. Implementation to the point of being a product is likely to require an industrial partnership, which of course is the purpose of this proposal. Currently, we have completed the overall specification and are implementing several of the critical modules (as described below).

6) Potential impact, applicability
This is the "container" for the rest of the IP. The GUI-based product can be built out the same components by adding a standard GUI.

2.3 Datapath Width Sizing/Precision Management

Context/Problem
Programmable circuits allow minimization of chip resources as a function of the computation. This allows chip resources to be used for other purposes, e.g. additional parallelism, additional computations, error checking, storage, etc. One minimization is in the number of bits in the datapath including ALUs, multipliers etc. from the 32 or 64 in a microprocessor down to something smaller, including sometimes a single bit.

The problem maximizing the reduction while minimizing the risk that the computation will be compromised.

State of the art
- FP work. Supercomputer applications people have worried about fitting and precision since the early days of computing.
- DSP stuff. Specifying bit widths is part of some tools such as SysGen.
The techniques include
- on the high end: worst case analysis, sampling from the expected domain input
- on the low end interval and ad-hoc analysis and ad-hoc analysis
Likely that some statistical techniques may have been applied as well.

Brief Description/Method
Analyze the arithmetic using probability density functions to represent ranges of possible values or to represent uncertainty due to quantization errors. This is a superset of worst-case analysis; it gives much more statistical information and so allow for substantially more efficiency. Gives similar kinds of results as sampling, but analytically, and so does not depend on all data being known a priori.

Reference – CAAD Lab Note TVC030630

Potential impact, applicability
Usable anywhere +-*/ arithmetic is used. Can accept input from sampling techniques. Is analytic, so it can be built into compilers as a form of constraint propagation. Gives developers clear view of amounts of uncertainty. Thresholds, e.g. for dropping LSBs are user-tunable and have clear statistical meaning. May be amenable to backwards constraint propagation — that needs more study.

Status (Implementation, verification)
Compile-time so no circuit needs to be implemented. Some code has been written to allow evaluation of the technique. It uses symbolic algebra for processing the PDFs, so it can give lots of results that might be hard to reach otherwise.

2.4 Management of Parallel Vector Access

Context/Problem
In microarray analysis, many important techniques are based on examining combinations of thousands of vectors. When the size of the combination is ¿ 2, the number of combinations is very large, although the number of actual vectors is quite manageable. In programmable circuits, it is possible to perform hundreds of the computations in parallel making these computations tractable. The problem is to orchestrate the routing of the vectors from the storage to the parallel computation units so that combinations are available for processing at least once, but rarely more than that.

State of the art
No related work known other than some basic mathematics in Design Theory and Steiner Systems.

Brief Description/Method
Hierarchical buffering. Routing network. Buffer loading and indexing with methods from combinatorics and design theory.

Reference – FPL03 paper [9]

Potential impact, applicability
Immediate impact is in enabling several items below. Further applicability possible, although not immediately apparent where. Problems that can use this approach have the following characteristics:
- Given some set of data, the solution evaluates all size-k subsets of that data.
- It is possible to evaluate many subsets concurrently.

Status (Implementation, verification)
Hardware implementation is complete. Compiler-level in progress.

2.5 Protein Docking Computation

Context/Problem
One of the most important problems in computational chemistry and molecular dynamics with applications in intelligent drug design is finding how and where molecules dock with proteins. The basic science is well understood: the issue is simply one of computational complexity.

State of the art
Very large body of work. No known results with FPGAs. Standard methods work through transforms.

Brief Description/Method
Use of direct correlation, rotation through coordinate axis transformation, coordinate axis transformation through indexing. Use of small-integer scoring functions, possibly scoring different features (e.g. core collisions versus surface interactions) separately using saturating counters. Sparse voxel models are typical. No use has been made yet of sparseness. May be a candidate for acceleration techniques Work in progress on using compressed data/hardware structures for the above computations.

References: CAAD Lab Notes TVC030508 and TVC030527

Potential impact, applicability
Obvious potential impact.

Status (Implementation, verification)
Paper design.

2.6 Detection of Tandem Repeats and Palindromes

Context/Problem
Genomics has created a vast database of sequences. One of the most useful features is subsequences that repeat some number of times (Tandem Repeats) or repeat backwards (Palindromes). Methods of interest involve both precise matches and matches with a small number of errors. Problems are computational complexity and creating flexible implementations that satisfy the large number of variations of these problems.

State of the art
PC research programs. No known FPGA work.

Brief Description/Method
The computational requirement is somewhat different than that of partial string matching. In the latter, a large number of replacements and insertions/deletions are possible, indicating DP methods. Here, direct use of cascaded comparators is preferable. Our implementation has novelty at the circuit design level.

Reference: Notebooks only.

Potential impact, applicability
Analysis should proceed faster than the 10G letters/second making this ideal for use as part of a high-end database analysis system.

Status (Implementation, verification)
Proof-of-concept modeling done previously. Implementation proceeding; to be completed in ¡ 1 month. Verification is trivial upon implementation.

2.7 Suffix-Tree-Based Algorithms

Context/Problem
Same context as (4). A large body of string problems that can be solved using algorithms based on suffix trees.

State of the art
No known FPGA implementation.

Brief Description/Method
Novelty is in recasting the algorithms for fine-grained massively parallel processing.

Reference: Notebooks only.

Potential impact, applicability
Analysis should proceed faster than the 10G letters/second making this ideal for use as part of a high-end database analysis system.

Status (Implementation, verification)
Implementation in progress, to be completed in ¡ 1 month. Verification is trivial upon implementation.

2.8 Certain Optimization Methods with Applications in Microarray Analysis

Context/Problem
Microarray analysis involves techniques different from standard statistical methods. This is especially true when microarrays are used to "reverse engineer" cell functions. In this case learning techniques such as Bayesian networks appear promising. The problem is that the computation of Bayesian nets has exponential complexity.

State of the art
No known FPGA implementation of Bayesian nets or related techniques.

Brief Description/Method
A method of computing Bayesian nets that appears to be suited for FPGA implementation is the AI optimization technique of hill-climbing. Our implementation involves a novel use of computational cells coupled with associative processing hardware.

Reference: CAAD Lab Note MCH020323

Status (Implementation, verification)
Paper design.

Potential impact, applicability
Reverse engineering molecular pathways was the original motivation for microarrays. A computational technique that enables progress in this area enables progress in solving one of the fundamental problem in all of biology. Hill-climbing is an important optimization technique with applicability far beyond microarrays. Also, many problems that are solved using other techniques can be recast as hill-climbing problems if there is a good implementation.

2.9 K-Combination Microarray Analysis Through Linear Regression

Context/Problem
From microarray data, find the K genes (K > 2) that best correlate with a particular outcome. The problem is that such computations can take weeks on a PC.

State of the art
Our work.

Brief Description/Method
Reference: FPL03 paper

Status (Implementation, verification)
Implemented and verified in Xilinx Tools.

Potential impact, applicability
This work is a point study but has proved valuable in proof-of-concept. Also, some of the techniques developed here have further applications. In particular: run-time validity checking, missing-value analysis, load-balancing between compute units at different speeds, resource balancing to make best use of hardware

resources (e.g. block multiply).

2.10 Microarray Analysis Techniques Using K-Combinations and Heuristics

Context/Problem
Microarray analysis involves techniques different from standard statistical methods. Many of these involve clustering, self-organization, etc. The problem is that these methods do not work as well as other methods. It is easy to construct examples where they result in outright misclassification. There are at least two reasons why other techniques have been avoided: one is a cultural adversity of methods that are not primarily statistical and the other is an understandable reluctance to implement methods of substantially greater computational complexity.

State of the art
Many techniques are based entirely on comparisons of all-pairs of vectors. Apparently none are based on larger sets.

Brief Description/Method
Take advantage of K-combination implementations in previous items. Novelty is in application of heuristic techniques in a domain currently dominated by statistical techniques. This appear promising because it has been found elsewhere (e.g. in image analysis) that heuristic-based techniques can dramatically outperform statistical techniques.

Reference: CAAD Lab Note MCH030618

Status (Implementation, verification)
Paper design.

Potential impact, applicability
Improved microarray analysis.

2.11 Environment and Structures for FPGA Implementation of a Variety of Microarray Analysis Techniques

Context/Problem
The segmentation techniques (described above) are extremely computationally intensive.

State of the art
None.

Brief Description/Method
Use various techniques in other items combined with novel optimization techniques.

Status (Implementation, verification)
Design in progress.

Potential impact, applicability
Improved microarray analysis.

2.12 Single Chip SIMD Architecture

Context/Problem
Massively parallel SIMD array computers at one time dominated bioinformatics computation. Their commercial demise has shifted them to the background, though the Kestrel project at UCSD is one example of their continued promise. The benefits of SIMD arrays are their extremely high computational capability and simplicity of programming. Continued advances in process technology enable colocation of controller, array, and even sensor on a single device, removing the major objection to their use. The problem is that new technologies bring new problems. By getting rid of the slack in the designs, problems have emerged in

balancing data working set instruction distribution.

State of the art
Our work.

Brief Description/Method
References: Numerous publications including [3, 2].

Status (Implementation, verification)
Prototypes have been implemented in Verilog, simulated, and synthesized.

Potential impact, applicability
This could be very effective. For example, there is already a history of success with 1990-era machines. Updating the technology could restore place. Less performance gain than the best direct FPGA implementations of applications, but much easier to program. Potentially well-suited to Gibbs sampling, a technique used for locating features common across many strings. Applicability also to domains beyond BCB, e.g. computer vision.

3 IP to be developed

Creating the entire product as outlined above is the work of several engineer-years. This will require commercial support through investment and/or partnership. Our goal is to attract such support and obtain the best share possible in the venture. The key is making the central IP, the language and the compiler, as valuable as possible. To do this we need to

- Show the broad usefulness of FPGAs for BCB, especially towards the emerging areas of BCB – and those with computational needs that have so far not been served by similar dedicated hardware.
- Show that programmers can create applications by mixing-and-matching functions or otherwise using the language.

We now describe tasks that can be reasonably completed over the next 12 to 18 months to improve the value and marketability of the entire IP package.

- Develop demos of selected applications. Those on which we have already made substantial progress are: tandem repeats and palindromes, part of BLAST, linear discriminators, and a convolution kernel for protein docking.
- Create selected modules. Some key modules are associative operators, dynamic programming automaton, and an objective function automaton.
- Finish language design. Specify the object architecture for the functions and templates.
- Integrate FPGA performance infrastructure into the compiler back-end. In other words, integrate the parallel unit optimize and the precision management tool into the Java to VHDL compiler.

4 Schedule and Milestones

In the previous section we outlined work to be done over the next 12 to 18 months:
10/2003 – string demos (tandem repeats and palindromes)
12/2003 – finish language design, partial BLAST demo
2/2004 – DP automaton module and template
4/2004 – convolution kernel demo
6/2004 – linear discriminator demo, finish several functions
8/2004 – finish back-end compiler integration

At this point, if we have not already done so, we should establish a commercial partnership or other funding.

References

[1] Frost and Sullivan. *Bioinformatics: Players, Problems, and Processes in Computational Biology.* San Antonio, TX, 2002.

[2] Herbordt, M. C., Cravy, J., and Lin, C. Memory considerations for high performance simd systems with on-chip control. In *Proc. of Computer Architectures for Machine Perception* (2003).

[3] Herbordt, M. C., Cravy, J., Zhang, H., and Lin, C. An array control unit for high-speed SIMD arrays. In *Proc. of Computer Architectures for Machine Perception* (2000), pp. 293–301.

[4] Kim, Y., Noh, M.-J., Han, T.-D., Kim, S.-D., and Yang, S.-B. Finding genes for cancer classification: Many genes and small number of samples. In *2nd Annual Houston Forum on Cancer Genomics and Informatics* (2001).

[5] Lander, E. Foreward. In *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins,* A. D. Baxevanis and B. F. F. Ouellette, Eds. Wiley, 2001.

[6] Lander, E. The human genome and beyond. Distinguished Lecture at Boston University, April 2003.

[7] Sun Microsystems. *Briefing in Computational Biology.* Palo Alto, CA, 2002.

[8] Time Logic Corp. *Various Press Releases.* www.timelogic.com, 2001.

[9] Van Court, T., Herbordt, M. C., and Barton, R. Case study of a functional genomics application for an fpga-based coprocessor. In *Proc. of Field Programmable Logic and Applications* (2003).

[10] Yamaguchi, Y., Miyajima, Y., Maruyama, T., and Konagaya, A. High speed homology search using run-time reconfiguration. In *Proc. of Field Programmable Logic and Applications* (2002), pp. 281–291.

Case Study of a Functional Genomics Application for an FPGA-Based Coprocessor*

Tom Van Court[1], Martin C. Herbordt[1], and Richard J. Barton[2]

[1] Department of Electrical and Computer Engineering
Boston University, Boston, MA 02215
herbordt|tvancour@bu.edu
[2] Department of Electrical and Computer Engineering
University of Houston, Houston, TX 77204
rbarton@uh.edu

Abstract. Although microarrays are already having a tremendous impact on biomedical science, they still present great computational challenges. We examine a particular problem involving the computation of linear regressions on a large number of vector combinations in a high-dimensional parameter space, a problem that was found to be virtually intractable on a PC cluster. We observe that characteristics of this problem map particularly well to FPGAs and confirm this with an implementation that results in a 1000-fold speed-up over a serial implementation. Other contributions involve details of the implementation, including an anlysis of the number of bits required at various points in the computation. Since this problem is representative of many in functional genomics, part of the overall significance of this work is that it points to a potential new area of applicability for FPGA coprocessors.

1 Introduction

Microarrays measure simultaneously the expression products of thousands of genes in a tissue sample and so are being used to investigate a number of critical biology questions. Among these are (paraphrasing from pages 19-20 of [3]): Given the effect of 5000 drugs on various cancer cell lines, which gene is most predictive of the responsiveness of the cell line to a particular chemotherapeutic agent? or Is there a group of genes that can serve to distinguish the outcomes of patients with disease *ijk* who are otherwise indistinguishable? or Which of all known genes have a pattern of expression similar to those genes regulated by factor *xyz*?

As exciting as this usage is, microarray analysis is extremely challenging. One issue is that the data are noisy and the noise is often difficult to characterize. Another issue is that the number of measured quantities is invariably much larger than the number of samples. This results in an underconstrained system not amenable to traditional statistical analysis such as finding correlations. As

* This work was supported in part by the National Science Foundation through CAREER award #9702483 and by a grant from the Compaq Computer Corporation.

a result of these and other difficulties, techniques are used (often derived from machine learning) that provide a focus of attention, or a visualization, from which biological significance can be inferred. Among these are various forms of: clustering, inference nets, and decision trees. Their computational complexity ranges from the trivial to the intractable. However, given the cost of obtaining microarray data, the fact that further biological interpretation is usually still required, and the value of many of the most rudimentary computations, most analysis applications are tailored to run fairly quickly on ordinary PCs.

It is undoubtedly the case however, that biologists would like to ask far more complex questions and that increased computational capability would increase their ability to answer them. With applications such as those listed above, however, even a slightly harder question can result in an increase by orders of magnitude in computational complexity. This is true of the problem we investigate here.

Kim [2] would like to find a set of genes whose expression can be used to determine whether liver tissue samples are metastatic or non-metastatic. For biological reasons, it is likely that three genes is an appropriate number to make this determination. Kim further proposes that use of linear regression would be appropriate to evaluate the gene subsets over the available samples. Since there are thousands of potential genes, 10^{10} to 10^{12} data subsets need to be processed. Although simple to implement, he reported that this computation was intractable even on his small cluster of PCs.

We decided to investigate the prospects of supporting this computation on an FPGA-based coprocessor. There are many reasons:

- It is an excellent match computationally. The data-set size, the amount of computation per datum, the nature of the individual computations, and the data-type size all are favorable to FPGAs.
- This computation is representative of a large number of similar computations in both microarray analysis as well as in broader bioinformatics and computational biology (BCB). Therefore, demonstrating the efficacy for this problem would have much wider significance. Note that Yamaguchi et al. [8] have shown similar success (to what we show here) for a different application in bioinformatics, but one that has substantially different computational characteristics.
- There are a large number—perhaps thousands—of potential users. Therefore a low-cost distributed solution is much more attractive than a centralized resource such as a large-scale cluster. This is especially true since (as we will show) it would take a very large cluster to match performance.
- The state of algorithmics in microarray analysis (and some other areas of bioinformatics) is one of flux. Therefore a solution based on a generic PC and coprocessor may be more attractive than hardwired alternatives such as ASICs. The same would be true with respect to turn-key software/hardware systems, such as those provided by a number of vendors, assuming that they provided solutions to these problems at all. Also, both of these "hardwired" alternatives are extremely expensive.

Although we are still refining our solutions, what we have found is that we can obtain a speed-up of a factor of more than 1000 over an optimized serial version running on a 1.7GHz Pentium IV PC. These results have so far been achieved only in simulation but we expect to have them running before the final version of this paper is submitted.

Our most basic contribution is the speed-up for this particular problem: a set of 10,000 genes can be examined in 10 minutes instead of 19 days. Other contributions have to do with the actual implementation, including an analysis of the number of bits required at various points in the computation. Finally, as this problem has similar characteristics to many others in microarray analysis, we show that there is potential for broader applicability of FPGA coprocessors in this very important domain.

The rest of this paper is organized as follows. In the next section we present the problem formally and describe the serial implementation. There follows a description of the FPGA design and implementation including an analysis datapath widths. We conclude with a discussion of some further applications.

2 Application Detail and Serial Implementation

The technique used involves computing linear regressions for all 3-way combinations of genes. Each gene is a (column) vector in the matrix X with length equal to the number of samples n; the solutions are the column vector Y also with length n. The goodness of fit is determined by computing the R^2 values of each regression.

Examining a standard statistics reference [5], we find that the estimators $\hat{\beta}_0, \ldots, \hat{\beta}_n$ comprising the column vector $\hat{\beta}$ can be computed as follows

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

where the first column of X consists additionally of a column of 1's. However, since much of computational complexity results from the inversion, it is important to reduce the rank of the matrix. This is done by centering the data, which results in a $\hat{\beta}= \hat{\beta}_1, \ldots, \hat{\beta}_n$ of

$$\hat{\beta} = [(X^+)^T X^+]^{-1} (+) X^T Y^+$$

and

$$R^2 = \frac{\hat{\beta}(\mathbf{X}^+)^{\mathbf{T}}\mathbf{Y}^+}{(\mathbf{Y}^+)^{\mathbf{T}}\mathbf{Y}^+}$$

where X^+ is the n x 3 matrix containing vectors with elements $X_i - \overline{X}$ and Y^+ is the n x 1 matrix containing the values of $Y_i - \overline{Y}$. Note that in centered mode we do not need to compute $\hat{\beta}_0$ and thus do not need the column of ones in X. We therefore operate on 3 x 3 matrices rather than 4 x 4.

The primary operations to be computed are 10 dot products of length n, 3 summations also of length n, and an inversion of a 3x3 matrix.

In our tests we used sample data derived from a human breast tumor study by Perou, et al. [4]. It is typical of that generated in microarray studies and consists of 9218 gene expression patterns (including controls) from 84 samples. The data are presented as ratios, but as is standard practice, we took the log, truncated, rounded, and normalized the data so that they are all integer values in the range of -4 to +4. Using four bits is on the high end of information per sample; often only two bits are used. The result vector consists of binary data (0/1 for diseased/healthy).

An important consideration in microarray analysis is dealing with missing data. That is, the microarray value for a gene/sample expression is sometimes completely unreadable; in that case no value at all is reported. For example, in the Perou data set, 4237 of 9218 rows (46%) contain missing data elements and 84.2% of all row triplets contain missing data. If missing data are not handled properly, they can dominate the regressions and render results meaningless. We take one of the simpler approaches: for a given combination of genes, for each sample with missing data, we eliminate that sample from consideration for all genes.

When implementing the algorithm, one quickly observes that each dot-product gets used a very large number of times. In fact, it makes sense to precompute and save *all* of the n x n dot products and then use them to compute the $n(n-1)(n-2)$ inversions in a succeeding phase. This eliminates roughly a factor of n dot products. Unfortunately, this does not account for missing data: each dot-product can have drop-outs not just from data missing from the two vectors being multiplied, but also from the third vector of the set. Since this third vector changes for every set, it follows that all dot products must be recomputed for every iteration.

We created two versions of the serial code. One was a mirror of the FPGA implementation and was used to verify results, especially with respect to maintaining precision. The other was used to generate timing and so was optimized for serial execution on a modern processor. We briefly describe some of these optimizations. At the coding level: (i) we unrolled the loops around the set size, although not the vector size, the latter already being optimized by the processor/compiler. Also, (ii) we stayed in fixed point arithmetic for as long as possible as is described presently. And (iii) we hardwired the matrix inversion. At the algebraic level, (i) we factored out the subtractions of the means from the inner loop so that they only need to be performed once for each sum of products. Also, (ii) for the matrix inversion in the R^2 computation, we do not immediately divide each cofactor by the determinant and so both reduce the number of computations make them ints rather than floats. One potential optimization that had little effect on performance was taking advantage of the small number of bits to reduce the data type size. This helped for neither fixed nor floating point arithmetic.

The R^2 for each set of genes was computed in 10.1us on a 1.7GHz Pentium IV PC. Because of the tremendous data locality, there was no drop-off in performance with respect to the number of gene combinations evaluated. This means

that evaluating 3-way combinations of 1,000 genes takes about half an hour, while 3-way combinations of 10,000 genes takes more than 19 days, and 20,000 genes takes more than five months. In one way these results are very impressive: the small number of microseconds was the time for a single instruction a generation ago; here it is thousands of instructions. Clearly L1 cache and available ILP are being used to a very high degree, the latter not surprisingly due to the numerous multiply-accumulate (MAC) computations and the hardwired invert.

Still, these results confirm our initial assumption: that this computation, while perhaps not "heroic" in the grand-challenge sense, is still outside the realm of usage in exploratory data analysis. For this and similar computations to be readily usable as part of an analysis toolbox, days need to be reduced to minutes.

3 FPGA Methods, Implementation, and Results

3.1 Description

Hardware is assumed to be an FPGA on a commercial PCI board plugged into a PC. As the amount of reuse per datum is very high, details about the particular board and interface do not have a significant impact on our results. Still, we anticipate having a working system by the time the final version of this paper is submitted. The rest of this discussion describes simulations in the Xilinx ISE 5.1i envionment [7] for Virtex-II Pro XC2VP125 gate array [6] and anticipates implementation on a generic coprocessor board. The Virtex-II Pro product family is especially interesting because of its large gate count, large on-chip memory, and specialized resources. Those resources include processor cores, dedicated multipliers, and available libraries of pre-programmed utility functions. Eventual implementations of this design may be forced to use an FPGA with fewer resources, so the design may be scaled down to match available hardware.

The FPGA logic design consists of three main components: system and memory interface, arithmetic processing, and on-chip memory control. We now describe these in turn.

The interface to the system bus and off-chip memory are not yet part of the system being simulated. However standard IP cores exist for these functions, including PCI interfaces and DDR memory chip interfaces. The particulars will be determined by the choice of co-processor board. In any case, the system interface is not critical to this application's performance as we now describe. On-chip memory includes over 500 RAM blocks totaling 10Mbits, or more than enough to hold all of the data (about 2.5Mbits) in our initial target application. This will even allow data to be replicated across multiple RAMs, allowing simultaneous access to a large fraction of the data without contention. Presence of many independently addressable RAM blocks ensures large total bandwidth between the on-chip memory and the arithmetic logic.

The on-chip memory is the first tier in the application's memory hierarchy. Although not needed for the current application, a second tier of memory is available on coprocessor boards, configured as RAM or as FIFO. A third tier

is host RAM, a fourth, host virtual memory. The initial design uses dedicated logic for accessing on-chip memory and host drivers for DMA to and from the coprocessor. The Virtex II Pro FPGAs, however, include dedicated PowerPC cores. Although these are not used in our initial design, they represent a major design resource and may be especially helpful in scheduling memory transfers between the chip, local memory, and host.

Arithmetic processing is divided into two major segments, the dot-products and summations (DPS), and the covariance matrix, inverse, and the regression results (CIR). Each DPS accepts four data vectors, three X values and one Y. The Y represents the diagnosis, 0 or 1 for cancerous or healthy samples respectively. The X values represent expression levels, encoded as four bit values. The encoding represents a symmetric range from $-N$ to $+N$. In four bits, that can be -7 to +7, biased to a 0-14 range of values. The remaining value (15, binary 1111) is a Not-a-Number (NaN) code that represents missing or invalid input data. The DPS unit, illustrated in Figure 1, consists of

- Counters, to tally valid data sets and Y values,
- Accumulators to total the X vectors and XY products, and
- MAC sections to total the $X_i X_j$ products.

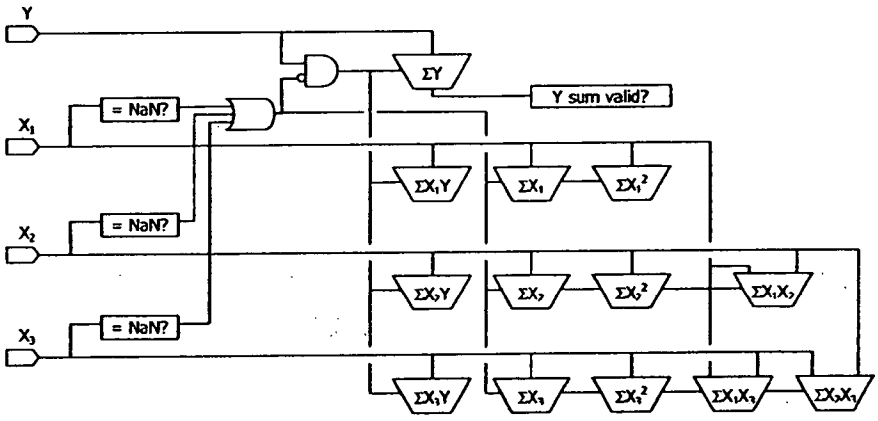


Fig. 1. Dot products and summations.

Also note the support for missing data. In Figure 1, the boxes labeled = NaN? detect missing data and propagate that fact to the MAC units where the accumulation of the invalid summands is blocked. If, for example $X_1[i]$ is a missing value, then the summation terms $X_1[i]^2$, $X_1[i]X_2[i]$, $X_1[i]X_3[i]$ and $X_1[i]Y[i]$ are meaningless. Taking a conservative approach, $X_2[i]X_3[i]$, $X_2[i]Y[i]$, $X_3[i]Y[i]$ are omitted from their respective sums.

This DPS computation takes advantage of the 1-bit Y values in several ways. First, the summation of Y values reduces from an accumulator to a counter.

Second, the summation of Y^2 values is redundant. Given 0 and 1 as the only possible Y values, the sums of Y and Y^2 are identical. Third, the XY sums are accumulators, conditionally adding X elements depending on Y being 0 or 1. Only the dot products of X vectors require true multiplication, 4 bit x 4 bit products of unsigned values, giving 8 bit results.

Once the whole vector is processed, the DPS results are latched for input to the CIR. DPS results consist of outputs from all the accumulators in Figure 1, a valid data counter, and the validity indicator (overflow and minimum-Y tests—details below). The result is presented broadside to the CIR section. That section consists entirely of unclocked combinational logic, including adders, subtracters, and multipliers. It first computes the 3x3 covariance matrix from the X dot products. That feeds a closed form inversion of the covariance matrix.

3.2 Optimizations and Safety Checks

An optimization fundamental to obtaining very high performance is to minimize the datapath at all points in the computation. Although worst-case calculations of the minimum datapath width are simple and guaranteed to be safe, they almost certainly result in a far more conservative implementation than necessary. It does not cost much to store the extra bits, but they have large effect on the time and chip resources needed to perform the computations.

We address this in two ways: (i) *a priori* determination of the maximum datapath widths and (ii) confirmation that overflow has not occurred. There are two aspects to datapath width determination: the bits that can be dropped at the high end (because they represent worst cases that never occur in practice) and bits that can be dropped at the low end (because the loss of precision is insignificant).

The *a priori* path width determination is done in two ways: empirical and theoretical. On the theoretical side, the need for less significant bits is estimated using interval arithmetic [1]. Precision is then verified empirically by sampling test data off-line. That is, covariance and inversion calculations are simulated on the samples. Then the covariance and inverse matrices are multiplied together. The ideal result would be an identity matrix; measured precision is acceptable if results lie within a specified distance of the ideal.

Also on the theoretical side, worst-case analysis sets an upper bound on the number of high-order bits required. Then the application data are again sampled off-line and the number of bits used at each step is measured. This gives statistics of actual bit usage. High-order bits are allocated to cover the samples observed, plus some margin based on observed standard deviations.

The implementation, then, handles all data values that can reasonably be expected. To be thorough, however, the FPGA logic checks for overflow at each step. The DPS and CIR stages both present validity indicators, stating whether erroneous calculations were detected at any point.

Our implementation also performs another safety check. As described above, invalid data (a NaN value) at some element in any one of the X vectors invalidates the entire row of the calculation. Given the frequency of invalid input data,

there is a chance that most or all values in a set will be discarded. In this application, the Y values represent the diagnosis, so they are critically important. For example, our data include only a small number of healthy samples among the total of 84 and so losses there can render a single regression meaningless. We have therefore implemented a configurable test that compares the Y total to a chosen threshold and invalidates the DPS result if the total does not meet the minimum.

Another optimization is the use of closed form inversion. Although this method has many problems when applied to large matrices, it works well in this case. First, the small matrix size keeps the matrix cofactors reasonable in magnitude. Second, the number of terms is small enough to prevent destabilizing numerical effects from contaminating the result. Third, since it is an unconditional sequence of arithmetic operations, there is no need for control logic or clocked latching of intermediate results. Fourth, some simplifications become possible since the covariance matrix is known to be symmetric. The inverse matrix is not computed in exact form, since that would require dividing each element by the determinant of the covariance matrix. Instead, the cofactors (numerators) and determinant (denominator) are kept separate. The matrix inverse is then used to compute the regression and correlation coefficients.

3.3 Timing

The propagation delay along the CIR logic path is computed statically, and a simple counter determines the time at which results are expected to be stable. When all arithmetic and validity results are believed stable, the CIR output is latched and the CIR is ready for a new set of input data.

Given our current data set and other implementation parameters, the CIR propagation delay is roughly 40ns. The DPS section can clock at 5ns, however. Given data vectors of length 80, the DPS requires 400ns to compute its result. The imbalance, 40ns vs. 400ns, is conspicuous. The other conspicuous imbalance is in the resources used by each section. Each CIR requires 48 multipliers, roughly 10% of the commited multiplication logic available. The DPS, however, uses only 4x4 products, which can be built effectively from random logic. The DPS uses smaller, less specialized logic resources, under 1% of the chip total. Considering both execution time and logic resources used, each CIR should serve about 10 DPS units to achieve maximum logic activity, assuming a vector length of 80. Figure 2 shows how this could be done.

These design values (vector length, DPS propagation delay, etc) are all parameters that follow from the specific problem at hand. Different data sets would represent different sets of design parameters, and would yield different specific results. The analysis would follow the same general pattern, however, over wide ranges of problem parameters.

The planned DPSs all start and end their computations at the same time, latch their results, and begin processing the next set of vectors. After latching the DPS results, separate logic presents results from each DPS to the CIR one by one. A counter (not shown in Figure 2) holds input stable for the CIR propagation

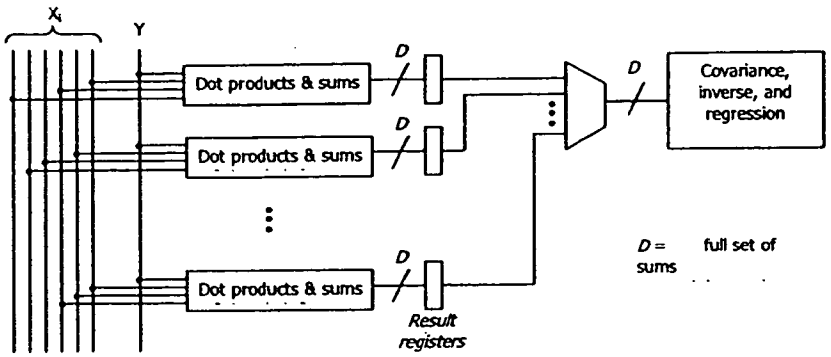


Fig. 2. Computation balancing.

delay, then stores the CIR result and selects the next latched DPS result. If the system is well balanced, there will be very little lead or lag time between saving the last CIR result and latching the next set of DPS results.

It is not necessary to run all DPSs with the same begin and end times on their respective sets of vectors. Synchronizing them, however, will simplify the interface to the on-chip memories. Synchronization will also allow DPSs to share inputs, minimizing the memory bandwidth load. Depending on the number of DPSs available across the whole chip, full utilization might require as few as 7 to 10 X vectors at a time. Taking all possible triples from seven inputs, there are $\binom{7}{3}$ or 35 combinations of X vectors available. For 8, 9, or 10 X vectors, there are 56, 84, or 120 distinct triples available. Vectors are loaded into the DPS element-by-element, meaning that as few as *40* X bits (per cycle) could keep the whole chip active. Since we plan roughly 100 DPS units, each with 3 X inputs (12 input bits per DPS), it will take 1200 X data bits per cycle to keep the DPSs active. Doing that with only 40 bits of data represents a dramatic savings in access cost. The saved bandwith might then be committed to other purposes: redundant storage to avoid access conflicts, data exchange with off-chip memory or host, PowerPC access to results, rate conversion (if needed) to match memory access rates and computation rates, etc.

3.4 Throughput

The basic design consists of a pipeline with a single CIR (40ns) and single DPS (400ns). This gives a speed-up of a factor of 25 over the serial PC version. The Virtex II Pro easily fits 10 of these units (the critical resource being the multipliers) increasing the speed-up to 250x. Each CIR can serve 10 DPSs reducing the cycle time of the original unit to 40ns and increasing the total speed-up to a factor of 2500x. We believe that, given the available resources (especially the

available data transfer bandwidth and the method of using it sketched above) that this last factor should be approachable.

4 Discussion and Extensions

The 1000-fold+ speed-up derived from our FPGA implementation achieves our goal of reducing the duration of this computation from days to minutes. This is done while keeping the system cost (hardware and IT support) low. It is therefore quite plausible that this and related techniques could indeed become part of a computational toolbox for functional genomics, broadening the types of inferences possible from microarray data.

In the introduction, we described why the potential for speed-up was so great for this application, even though it already can use a serial processor at close to maximum capacity. Here we present those points in a slightly different way and state that a large number of BCB problems have a similar characteristic structure. In particular, they have:

- A high-dimensional parameter set that must be searched or enumerated to find an optimal solution.
- Simple performance criteria (score functions) derived from the parameters.
- A decomposable search strategy and/or score function.
- A large but manageable sample data set that must be processed to evaluate the score function for each candidate element in the parameter set.

A necessary extension to this work is achieving some generality in implementation: it is certainly expensive to buy a large cluster and hire a parallel applications programmer; it is perhaps even more problematic to find good FPGA designers. We are currently working in this direction.

References

1. Hansen, E., et al.: Topics in Interval Analysis. Clarendon Press, Oxford, U.K. (1969)
2. Kim, S.: Finding Genes for Cancer Classification: Many Genes and Small Number of Samples. 2nd Annual Houston Forum on Cancer Genomics and Informatics (2001)
3. Kohane, I.S., Kho, A.T., Butte, A.J.: Microarrays for an Integrative Genomics. MIT Press, Cambridge, MA (2003)
4. Perou, C.M., et al.: Molecular Portraits of Human Breast Tumors. Nature **406** (2000) 747-752
5. Ryan, T.P.: Modern Regression Methods. John Wiley and Sons, Inc., New York (1997)
6. Xilinx, Inc.: Virtex-II Pro Platform FPGA User Guide. (2002)
7. Xilinx, Inc.: Integrated Software Environment. (2002)
8. Yamaguchi, Y., Miyajima, Y, Maruyama, T., Konagaya, A.: High Speed Homology Search Using Run-Time Reconfiguration. In Proceedings of the 12th International Conference on Field Programmable Logic and Applications (2002) 281-291

Accelerating microarray analysis computations

Why are we looking at microarray computations? Microarrays are important and improving analysis has potential for impact.

Problem: current techniques appear satisfactory. For example not generally listed as a computational problem by people who do that sort of thing.

However: there are two types of computations, production and research. Production are straight-forward, relatively effective, and quick. There are therefore two opportunities: the tough researchy computations such as bayesian nets and enabling the increasing complexity of production computations.

Production: Dendrograms, k-sets, other clustering

Research: Bayesian nets

Dendrograms

Rationale

Dendrograms are perhaps the primary method of representing sets of microarray data. Dendrogram computation is trivial on a PC, taking on the order of minutes. Generally, more complex computations are not done. Must check, but I believe this is NOT because more complex computations have been systematically tried and found to be not worth it. Rather the following are likely: data are problematic so analysts generally do not think that heroic effort is worth it.

However, although the data are problematic, they are still very expensive to obtain. Also, we are starting to build libraries

We believe that if given the computi
-- not much

It is an open
question whether
-- complex computations

Standard dendrogram algorithm:

FOR ALL VECTOR PAIRS
- compute distance

UNTIL ONLY ONE NODE IS LEFT
- find best correlation among all pairs of leaves/nodes
- remove pair of leaves/nodes
- compute new node from pair of leaves/nodes
- find distance from new leaves/nodes to all other leaves/nodes

UNTIL DISPLAY IS FINISHED, DO DFS ON A PARAMETER

- output next

%%

Problems with normal dendrogram algorithm: do not use information as well as you could. Purpose is to visually correlate clusters. Much flexibility in choice of cluster and visualization criteria.

Why does it not work as well as it could?
(1) only pairs are used
(2) nodes are characterized by their centroid

#1 #1 without #2 might not be a problem by itself because of transitivity: If three vectors are close, then should be plotted with close links and short branches. Visually it will be very similar to triple. But then again it may be trouble (see example below). You can have data sets where you just keep adding vectors to a cluster because you have a match close to an individual point already in the cluster, thus susceptible to runaway cluster growth. Example: cluster with small but non-uniform spacing surrounded by cloud of points with much less density, but also non-uniform spacing. You may need to relax the distance criteria to get all the points on the inside, but then might get "tails" of points on the outside. One way to mitigate this is to use other merge criteria such as an analogs to common surface area and enclosure.

#2 How this behaves:

- If similarity is Euclidian distance, then forming a cluster from two vectors moves the centroid of this new node away from all other new candidates to join the cluster (otherwise they would have been closer and formed the cluster).

- An immediate problem: A third vector that was the next-best-match to one of the two members might now be a best match to a different vector. This cluster might then "walk away" from the previous cluster leaving the third vector to appear unrelated.

- What happens when a third vector is added to the cluster? The centroid will then move back "across" one of the previous vector (2D view). But in general, the more vectors are added, the harder it is to make a correspondence with a new vector. This is not NECESSARILY bad: these "left-over" points that are closer to a cluster rather than pairing with another vector really are not that close to the other vector.

- A bad case? Let's say we have a three nearly equidistant points. Two must form a cluster first. Pretty clearly, can build bad cases.

```
x x x  x x x
 y  x  x x x
 y  x  x  y
 y    y   y
 y      y
```

- How bad is it? Maybe not so bad. The "only pairs" constraint prevents run-away cluster building. The fact that clusters form a hierarchy means that we can create clusters "in the eye of the beholder," i.e. by looking at how multiple levels merge.

%%%

Clusters based on triples instead of doubles, e.g. to start.

The main data structure for the original algorithm is a tableau of all pairs. All triples would obviously be rediculous. Therefore, need to compute using only a small number of passes on triples and higher. Large number of ways to do this. Could

- Keep list of some number of highest matches, highest disjoint matches, etc. and recompute when a new batch is needed.
- Could use initial pass to get all triples, then use an exchange process to adjust.

%%%

Could "grow" triples and higher order clusters. Similar to segmentation, but keep basic dendro structure. If a vector is added to a pair with some threshold goodness, make the pair a triple rather than adding a node.

%%%

Segmentation

- Histogramming (region splitting). Find histogram peaks and create regions around them. In one way microarray gene clustering is a better domain for this than image segmentation by pixel intensity: the position is already that which gets segmented. So don't have problem of non-contiguous, similar-intensity pixels forming clusters. The disadvantage is that the vector space is very large, so creating a finite # of histogram bins is problematic. For 70 dimensions, creating even two bins per dimension is a problem!

- Region growing/merging. Merge regions if they meet some criteria. If we want to keep the dendrogram-like representation then need to keep hierarchy information. Characteristics of regions are position, boundary position, and shape. Shape in a high dimensional space is again problematic.
  * Shape may not be that bad: although each region is potentially a 70 dimensional hyperplane, it is one that is relatively straight-forward to find the boundaries of. Must be convex. New points could even {\em already be in the hyperplane}. Or could tell if close to a boundary outlier (cape) but not the center-of-mass. Or could do center of gravity.

- Both splitting and merging allow rating along multiple criteria.

- Might be difficult to tune without good biological knowledge.

- Allows you to not classify at all if spacing is to big. Or to classify in a component of "far apart."

%%%

Successive refinement

... through random exchange of vectors, simulated annealing, etc.

Helps if you have a well defined metric for cluster goodness! Of course can run multiple iterations with user-supplied feedback.

%%%%%%%%%%%%%%%%%%%%%%%%%%%% %%%%%%%%%%%%%%%%%

The most basic problem in hardware acceleration of BCB algorithms may be the semantic gap between biological reasoning and hardware design. Effective hardware acceleration generally seems to require some rephrasing of the biological problem, using terms that do not come naturally to biological scientists. The basic job of a language designer, then, is to encapsulate knowledge of hardware design, including novel circuits specific to BCB problems, in linguistic terms natural to BCB programmers.

Language design will follow contemporary object-oriented philosophy. Major features of the new language (TNL) will include:

-- Syntactic similarity to Java,
-- Class libraries for application support,
-- Modern software design techniques, and
-- Support for current best practices in application design.

Several advantages follow from using a familiar language as the basis for development. For the application writer, a familiar language reduces the learning effort and inefficiencies due to misuse of unfamiliar programming constructs. For the compiler developer, use of an existing language allows use of existing compiler technology as a base. This means the devleopers can take advantage of existing tools, optimizations, and implementations without having to master the whole field of compiler technology. Finally, existing languages are the result of many years of design experience, both good and bad. Use of an existing language avoids repetition of mistakes that have already been made and fixed.

Java, in particular, offers specific advantages as a base language. It is syntactically and semantically much simpler than (for example) C++. TNL will inevitably impose hardware-oriented constraints. Since Java is a simpler language, it has fewer features to constrain, so TNL will stay more true to its parent language. Other "C-like" hardware languages (e.g. SA-C) have been criticized for making so many changes, both additions and deletions, that they are really quite unlike the the language from which they claim derivation. Java appears to offer a base requiring fewer changes than C or C++, so TNL should avoid many such criticisms.

Java (unlike C or C++) omits explicit pointers. Since pointers are difficult to implement in FPGA logic, it is common for hardware languages to omit or constrain pointers. Pointers are central to normal C programming, so their omission creates a very un-C-like pattern of language usage. Since Java already omits explicit pointers, any pointer-related constraints on TNL will have much less effect on the language and on its users.

Java is an object-oriented language, so class libraries are the normal extension mechanism. Careful design of supporting class libraries should avoid the need for new (incompatible) language constructs in many cases. For example, SA-C introduces new loop constructs specific to common image processing tasks. The SA-C application is written in terms of the new constructs, and the application-specific logic appears as the loop body. The same task, in an object-oriented (OO) language, can be carried out using standard OO programming. The new loop construct would be phrased as a class in the support library. An application writer would create new object types inherited from that class, and application-specific logic would appear as normal over-rides of

methods (functions) in that class definition. Standard compiler technology can then integrate the loop body methods into the iteration logic, perhaps with agressive inlining and replication for parallelism.

Note that application writers can extend library classes without explicit reference to the definition of the parent class's body. This lets library classes be defined using primitives that are not available to the application writer. The library class implementations can use high-efficiency hardware constructs not available in the language, without requiring extensions to the language, much the way Java libraries can be defined in terms of C or assembly code using capabilities missing from the Java language.

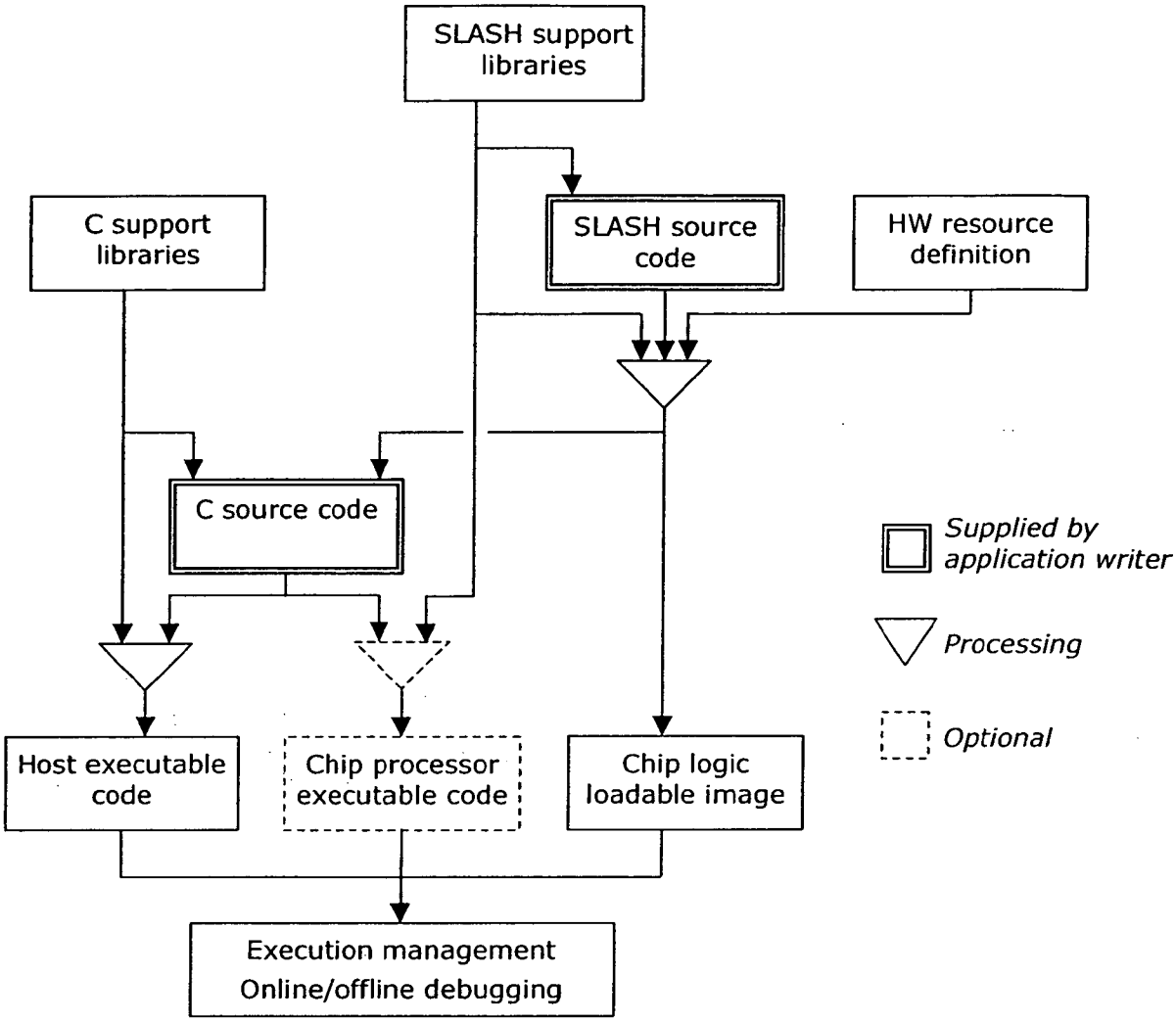
Also note that hardware parallelism comes from replication of functions into many units that execute concurrently. Logic replication is a natural feature of OO languages, since normal object types can be instantiated many times. There is a natural mapping between object instances and hardware replcias. Language scoping rules make parallelism easier, since they help prevent unintended interactions between object instances. That reduces the chance of error in handling many objects in parallel.

By staying close to standard Java, TNL encourages design using the best accepted practices for normal application development. Design Patterns (ref Gamma et al), for example, embody hard-won development expertise in clear development guidelines. Class extensions, described above, match the familiar 'Template Method' design pattern. The 'Strategy' design pattern represents another way for appication writers to encapsulate application logic and present it to generic system libraries, somewhat the way the C-language sort () function accepts a comparison operation as an input parameter. 'Iterator' and 'Visitor' design patterns can be adapted to hardware, since they proivide familiar techniques for traversing large or complex bodies of data.

Associative processing can also be phrased in OO terms. The heart of associative processing is some large number of data object instances, all evaluating their results in parallel. As described above, this seems to be a good match for an OO hardware language. Numbers of objects can easily be parameterized, to adapt to application requirements or available hardware resources.

Finally, system integration should be relatively straightforward when a familiar language describes the FPGA hardware implementation. FPGA coprocessors are only part of a PC-based system, and modern FPGA include CPU cores on chip. The host, on-chip, and possible other processors create a significant problem in integrating the software and hardware subsystems. Such integration is well outside the expertise of most application programmers. If the hardware is described in familiar software-based terms, however, it should be feasible to automate data movement between the different subsystems. Since the application writer uses similar language in all parts of the system, integration problems should be minimized.

Ideally, the exact dividing line between CPU and FPGA execution should be invisible to the application writer. That would allow the greatest flexibility in partitioning the problem across available computation resources. It would also allow alternative implementations of any application. The application writer would be able to select a software-only implementation with high debugging visibility or a hardware-intensive implemtation with high performance.



The diagram above illustrates the design flow used for implementing a program using this development environment. Translation tools (e.g. compilers) are not shown. The design flow includes the following components:

- **C source code.** These source files define the control and file IO portions of the application, as well as processing too complex for hardware implementation. Some part of the C code may be built for the host, others for on-chip processing (when available).

  Requires: Application content, generic header files from the support libraries, application-specific header files generated from the SLASH code.

  Generates: Linkable object code representing the application logic, including fragments generated from the SLASH code. Linkable code fragments for execution on the on-chip processor.

- **C support libraries.** These are provided as part of the application programming environment.

  Generates: Generic host software header files and linkable object modules containing generic hardware access features. Includes object modules specific to FPGA board and chip. Conditionally includes instrumentation for debugging and performance analysis.

- **Slash source code.** These source files specify the part of the application to be implemented in FGPA logic.

  Requires: Application content, generic header files from the support libraries.

  Generates: Host code, including header files and executable code representing application-specific access to the on-chip logic. May also generate interface definitions for coupling the on-chip processor to the on-chip application logic.

- **Slash support libraries.** These are provided as part of the application programming environment.

  Generates: Generic header files for SLASH code, generic support elements for logic synthesis, generic support code for on-chip processor[s]. Includes elements specific to the FPGA board and chip. Conditionally includes elements for debugging and performance analysis.

- **Hardware resource definition.** Within a family of FPGA chips and boards, specifies resources and technology dependencies needed for synthesis.

  Generates: Hardware resource constraints for hardware synthesis.

- **Host executable code.** Host portion of the application.

- **Chip processor code.** Portion of the application to be run on the on-chip processor.

- **Chip logic loadable image.** Portion of the application to be implemented in FPGA logic, in loadable form.

- **Execution management.** Coordinates loading and execution of host, on-chip processor, and FPGA logic. Captures interactive and non-interactive (dump-based) debug information.

Work items:

1 **Sample problem identification** –

We've already done substantial work here in identifying three overall domains. We define a domain as being a domain of biological research and as having certain basic computational e.g. in choice of key abstractions. Some domains have multiple subdomains as defined by computational characteristics, e.g. in string matching there are computations that are best done with dynamic programming, others best done with creation of suffix trees.

Substantial work will continue in finding new domains and finding more areas within each one of these. The character of the work to be done here is to identify applications that people care about, are computationally intensive, and appear to map well to FPGAs. Much of this work will consist of talking with practitioners, in addition to literature survey and some prototyping. One thing to be careful about is to watch for opportunities where it may be possible that new algorithms could be created with our new technology.

That is, algorithms or methods that have been viewed as promising, but largely ignored because of computational complexity. The bullets listed below signify ... [what we've found, versus work to be done]

- 1.1 String matching & variants
  - 1.1.1. Specific matching tasks
  - 1.1.2. Specific matching algorithms
- 1.2 Microarray data analysis
  - 1.2.1. Processing k-sets, e.g. with linear regression
  - 1.2.2. Pathway and Regulation network analysis – Bayesian nets
  - 1.2.3. Clustering
- 1.3 Molecular modeling – rational drug design
  - 1.3.1. Protein docking
- 1.4 New apps.

2 **Problem component characterization** – Prototyping

gain, we've done a bit here already. The character of the work is to find essential features at many different levels, especially which are interesting for FPGAs. For example, characteristics could include identifying algorithm kernels (e.g. matching), data structures (e.g. suffix tree), collective operation set (e.g. associative operations, vector operations), data types (2 bit, fixed point), data and working set sizes, etc. The bullets listed below signify ... [what we've found already versus work to be done]

- 2.1 Statistics
- 2.2 String matching
- 2.3 Linear algebra
- 2.4 Convolution
- 2.5 Machine arithmetic

2.5.1. Precision management
2.5.2. Missing values
2.5.3. Rational arithmetic
2.6 Used to specify logic cores & language primitives

3 **Hardware characterization** –

Some of this is simple enumeration, especially for use in the optimization stuff that we're already anticipating. Examples are number of LUTs, multipliers, etc. Some is qualitatively different, e.g. use of on-chip busses and processors. We are not likely to make use of this now. A third area is system-essential characterization, such as the board/processor interface.

4 **Logic core definitions, Function library** –

This includes all the "above language" constructs, both in HW and SW, including cell libraries and function libraries. It's difficult at this point to determine what will go into the libraries and what will become a new construct in the language. For example, the loop construct business is basically a function that has been munged into language syntax.

Certainly a large part of what we do will involve creation of a parameterized core library. Some of the cores will be obvious instantiations from (2); others will be "universally" useful components with high-probability of utility for our projected user base. A third kind will be nitty-gritty components such as PCI interfaces, memory controllers, etc.

4.1 Application derived primitives
4.1.1. Dynamic programming cell
4.1.2. Combinatoric staged memory
4.1.3. Suffix trees and suffix tree algorithms
4.1.4. Tandem-repeat and other matching structures
4.2 General purpose primitives
4.2.1. Associative operations among units – broadcast, leader election, reduction, content-based retrieval
4.2.2. Generalized pico-processor
4.2.3. Systolic operations
4.2.4. Linear algebra
4.2.5. SIMD processing
4.3 Application templates
4.3.1. Hill-climbing
4.3.2. Dynamic programming
4.4 Language-like primitives
4.4.1. Outer loops (Like SA-C)
4.4.2. Combinatorial loops, convolution, ...

4.5 Chip/board environment (maybe vendor-provided)
- 4.5.1. Porting layer
- 4.5.2. PCI interface
- 4.5.3. memory controllers

Besides the logic cores and language libraries, which might be considered programming interfaces themselves, at least the following will be provided: programming language, debugging interface, and GUI. See discussion at top of document for more on what we mean by programming language: there are innumerable issues with partitioning HW/SW, frameworks for dual specification, etc. I think that we need to avoid as much as possible the complexities inherent in the general model HW/SW model, unless we can pirate SystemC to our purpose. Still, we need, besides SLASH (the language that compiles to VHDL), a way to specify an interface to the FPGA for transfer of data back and forth if nothing else.

5 **Programming interfaces I – Programming language** –

This is envisioned to be a C/C++ variant a la SA-C.

5.1 Overall – Investigate public domain tools
- 5.1.1. SUIF (certainly useable as a compiler base if nothing else is available)
- 5.1.2. Structural Verilog (what's this called?)
- 5.1.3. SA-C compiler (public??)
- 5.1.4. Other?

5.2 Front end (syntax dependent)
- 5.2.1. Application language design
  - 5.2.1.1. Data types (NaNs, etc)
  - 5.2.1.2. SA-C style loops
  - 5.2.1.3. Vector primitives (sum, dot, max, ...)
- 5.2.2. Hardware feature language design (define available resources)
  - 5.2.2.1. Hardware specification language syntax (KISS)
- 5.2.3. Function partitioning (host, on-chip processor, logic)
- 5.2.4. Interface to host programming languages
- 5.2.5. Interface to hardware programming languages (e.g. Verilog modules)
- 5.2.6. Manual precision control – data types, precision issues with ops (e.g. which bits of a result get kept).

5.3 Intermediate-level processing (syntax & mostly technology independent)
- 5.3.1. Standard loop optimization (loop unrolling, strip mining, ...)
- 5.3.2. Hardware resource tradeoffs (e.g. hard vs. logic multipliers)
  - 5.3.2.1. Some hardware dependencies – parameterized
  - 5.3.2.2. Requires total amount of available resource
- 5.3.3. Static timing estimates (t and rate matching

5.3.4. Function partitioning (host, on-chip proc, logic)
5.3.5. Processing basics (error handling, ...)
5.3.6. Automated precision control
5.3.7. Host vs. board vs. chip memory allocation
5.3.8. On-chip memory staging and result sequencing (swap-in, swap-out, on-chip vs. on-board memory)

5.4 Back end (technology dependent)
5.4.1. Logic implementation
5.4.2. Arithmetic implementation & overflow detection
5.4.3. Debug instrumentation
5.4.4. Processor/logic interfaces
5.4.5. Host/chip synchronization and memory staging
5.4.6. Interfaces to standard C code

5.5 Host to chip porting layer
5.5.1. Isolation interface between host code and board driver
5.5.2. Host application to FPGA application interfaces
5.5.3. Control over FPGA logic pattern (reloading for phase 2, 3, ...)

5.6 Chip to host porting layer
5.6.1. Isolation interface between FPGA application and hardware environment
5.6.2. FPGA to host application interfaces

5.7 User documentation

6 **Programming Interfaces II – debugging** –

6.1 Interactive debugging
6.2 Dump-based debugging

7 **Programming Interfaces III – Non-textual interfaces** –

An interface like the Xilinx System Generator for DSP. To do this correctly requires massive effort in areas not critical to the research aspects of this project. However,

- need to make sure that what we come up with elsewhere is compatible with typical GUI design
- We may be able to get most of this in the public domain
- We may get help doing this from SI
- It would be a good project for non-PhD labor, e.g. masters thesis, undergraduate summer project, Sr. project, etc.

7.1 Spreadsheet, GUI, ...

Motivation and related work

Custom hardware for performing arithmetic generally uses fixed-point arithmetic: numbers with set bit-widths above and below the decimal point, and sometime with a fixed and implicit power-of-two multiplier. This representation raises three issues in hardware design:

1. Allocating enough bits to hold the full range of expected values,
2. Allocating enough bits to represent adequate precision in a computation, and
3. Allocating few enough bits that the hardware design meets its goals for gate count and performance.

Considerations #1 and #2 generally argue for large numbers of bits in the fixed-point numbers; number #3 argues for smaller numbers of bits. Loosely speaking, # 1 concerns the most significant bits in the calculation and #2 concerns the least significant bits. Somewhat different techniques are normally used to handle the most and least significant bits. These have fallen into two general categories: analytic or worst-case analysis, and empirical or statistical analysis.

Worst-case allocation

The simplest assumption for allocating significant bits is to use enough to handle any possible computation. Here, it is easy to define a few very conservative rules, for example:

a = b * c	bits(a) = bits(b) + bits(c)
a = b + c	bits(a) = max(bits(b), bits(c)) + 1
...	

At first glance, it seems hard to argue with the scheme. Variations exist, however. For example, one form of multiplication combines signed and unsigned operands – that may need an extra bit to represent the full range of possible signed results.

Variations also exist for the addition rule. As given, it acknowledges that the result may generate a carry out of the most significant bit. Some authors, however, constrain the sum to have no more than its operands' number of bits.

A minor variation on worst-case analysis uses not the number of bits in the operands, but the operand's ranges of values. This follows naturally from the Ada, VHDL, and other languages that state range values explicitly:

```
subtype DIGIT is natural range 0 to 9;
variable DIGIT b, c, d;
...
a1 := b * c * d;          -- 0 ≤ a1 ≤ 729
a2 := b + c + d;          -- 0 ≤ a2 ≤ 27
```

This example shows an additional advantage of analysis by value ranges rather than bit widths. Variables `b`, `c`, and `d` all require four-bit allocations. Worst-case analysis based on bit widths would require 12 bits for value `a1`, but worst-case based on value ranges requires only 10 bits. Bit-width analysis might allocate 6 bits for `a2` using the addition

rule given above; value range analysis shows that 5 bits are adequate. Value range analysis never allocates more bits than bit-width analysis, and can allocate significantly fewer.

Over-allocation at an early step of a calculation can cause problems later in that calculation. Suppose, for example, that a designer assume a 12-bit data path for a 10-bit value. Later in the calculation, the designer may decide to truncate some LSBs, because of error propagation in the LSBs and for economy of implementation. Right-truncation of 10-bit values in 12-bit data paths will remove more significance than the designer might realize – e.g. droppping four bits will leave six significant bits, not eight.

Still, worst-case analysis using bit-widths or value ranges has several attractive properties:

- It is analytic – it is easy to automate.
- It is conservative – it covers every conceivable operation.

Since worst-case analysis can operate automatically on the source code of a design, it has been used in a number of synthesis tools as part of *constraint propagation*. This is a technique for deriving resource needs for some variables from the resource needs of others. *Forward constraint propagation* works as shown in the examples – it derives the bit requirements of results from the bit requirements of operands.

Worst-case analysis can also be applied to *backward constraint propagation*, in which the operand requirements are derived from knowledge about the result. Suppose, for example that the indexing expression `x[j]` appears in some design, and that the `x` array is known to have 128 elements. Knowing the range of `x` indices, one can safely say that `j` need only have seven bits to handle this expression.

Empirical allocation

The empirical approach typically instruments a C/C++ program to make measurements of significant bits at every arithmetic operation. The program is then run on some amount of representative data. At the end of the run, statistical information is reported describing each of the instrumented operations. Instrumentation varies, but may include mean data values with standard deviation, min and max values encountered, and so on. The results are then used to guide bit-width at each point in the calculation, either automatically or manually. This has the clear advantage of being based on actual application data. It is also very good at capturing subtle behaviors. For example, some subtraction operations, given operands that are correlated and close in value, tend to produce results with fewer significant bits than either operand.

This technique also can also relax the constraint that every imaginable circumstance be handled precisely. The designer may choose to handle the rarest cases as errors or as special (e.g. saturated) values. Measured frequencies estimate the probabilities of extreme values, and let the designer quantify terms like "rare".

The empirical approach does, however, have significant disadvantages:

- It is difficult to draw inferences about any parts of the application except the ones instrumented. In particular, this approach can not represent relationships between dependent data items (e.g. operands and results).
- It is also constrained to representing only the test data used.
- It generally makes assumptions about the distribution of values that result from any calculation, e.g. a Gaussian distribution. This can yield misleading results, since calculations that alternate additions, multiplications, etc. do not meet the requirements of the Central Limit Theorem.

Error propagation

Discrete fixed-point numbers typically represent analog values derived from the real world. That means fixed-point values have some built-in error, typically ±½ of a unit in the lowest position (ULP). Even in exact calculations, the original quantization errors can propagate until they dominate a calculation.

Consider, for example, a sum of 1024 four-bit values. Even if every fixed-point value is the best possible representation of the analog real-world value, the errors could conceivably accumulate to ±½×1024, or ±512 ULPs away from the "true" answer. In that worst case, the 9 least significant bits of the sum would be contaminated by input quantization error. The grand total has 14 bits, but up to 9 of the least significant bits (LSBs)– more than half – have debatable accuracy. Clearly, the client of this value can operate on a good bit less than the 14 bits provided without losing any real accuracy. It is not immediately obvious, however, how many bits can be dropped without sacrificing accuracy.

Interval arithmetic provides a formalism for analyzing worst-case propagation of errors. Rather than representing numbers as single values a, interval arithmetic describes each number as a range of possible values, $[a_{min}, a_{max}]$. Alternative representations exist, including $a_{nom} \pm a_{err}$, capturing the nominal value and possible error in that value. This representation has been shown to have the same expressive power as value ranges, though.

As the name suggests, interval arithmetic redefines operators so as to operate on these ranges. Operators in interval arithmetic may be defined as follows:

$$[a_{min}, a_{max}] + [b_{min}, b_{max}] = [a_{min} + b_{min}, a_{max} + b_{max}]$$

$$[a_{min}, a_{max}] - [b_{min}, b_{max}] = [a_{min} - b_{max}, a_{max} - b_{min}]$$

$$[a_{min}, a_{max}] \times [b_{min}, b_{max}] = [a_{min} \times b_{min}, a_{max} \times b_{min}]$$

$$[a_{min}, a_{max}] / [b_{min}, b_{max}] = [a_{min} / b_{max}, a_{max} / b_{min}]$$

and so on. For simplicity, the rules shown omit handling of negative numbers – those can be added to the system with only small effort.

Interval analysis is another form of worst-case analysis. Instead of operating on the most significant bits of a calculation, however, it is most commonly used to analyze the LSBs, i.e. the bits subject to contamination by quantization error. Like traditional worst-case

analysis, interval analysis is completely analytic and amenable to automation. One can imagine synthesis tools using interval arithmetic to advise designers about possible losses of significance. Although some designers to have tools for interval analysis, the technique seems not to be wide-spread as a tool for hardware design.

Current state of the art in error propagation appears to rely on gut feel and some informal calculations. In that sum of 1024 values, for example, a designer could invoke the Central Limit Theorem (CLT) to assume that the computed sum some has a Gaussian distribution around the "true" value. Treating each input value as a uniform random variable on the interval [x-½ULP, x+½ULP], the variance σ of each value is 1/12 ULP. The variance of the sum, then is 1024×1/12, or roughly 80. In other words, the computed value could be wrong in its six least significant bits without exceeding the variance computed for the sum – perhaps only the eight most significant bits (MSBs) of the sum matter to this calculation.

The CLT applies only to sums, however. Typical calculations consist of some alternation of sums and products, violating the assumptions of the CLT. One could phrase continued products as sums of logarithms, then apply the CLT in a log-normal way. The alternation of arithmetic operations, however, limits the scope of this approach.

Summary

Current approaches to allocating fixed-point bits appear unsatisfactory for a number of reasons. Worst-case analysis on the raw data gives upper limits on the number of bits that could possibly be required for a calculation. It can easily be automated in terms of the source code for a design, for both forward and backward constraint propagation. It does not have any way to address the typical or most common cases, however. Empirical analysis of an algorithm can show the range of values at each step in a computation, given some input data set. Its results are constrained by the data used in the experiment, and does not offer any way to automate analysis.

Neither technique offers a satisfactory way to analyze the number of LSBs that can be dropped because of error propagation. Interval analysis gives some insight, but grossly overestimates the number of result bits affected by input errors. Rules of thumb can help estimate loss of significance in simple cases, but break down quickly in complex calculations.

The goal, then, is to develop an approach to bit allocation that combines the strengths of each approach in allocating bits to fixed point calculations. In particular, the approach:

- should be analytic, i.e. it should be amenable to automation,
- should be general enough to handle any ordinary arithmetic operations in a straightforward way,
- should be equally suitable for allocating the MSBs in a calculation and for eliminating LSBs that have been completely dominated by errors,
- should be offer adjustable thresholds of probabilities for extremes of range and degree of error contamination in a value, and
- should be compatible with traditional approaches, to the point of subsuming them in its analysis model.

The remainder of this note describes an approach that appears to meet all of these goals.

PDF-valued variables and expressions

This model treats data as random variables. It statically analyzes a program by treating values as probability density functions (PDFs), combined using ordinary rules of symbolic algebra. This is a natural outgrowth of interval analysis, where an interval may be treated as a random variable with a uniform distribution. This also matches the kind of reasoning used in empirical statistical analysis of measured frequencies. This model operates on probabilities rather than measured frequencies, but opens the way for some of the probability distributions to be based on empirical frequencies. Finally this approach gives a strong theoretical justification to the process of dropping error-ridden LSBs.

An input value, x_0, is generally a finite-precision approximation to some real-world value. Quantization error represents the discrepancy between the actual value and the digitized representation. Given only the digitized value, x_0, it is not possible to recreate the original value exactly. The digitized value could represent any real value within ½ ULP of x_0. This can be represented as a PDF, the probability of a real-world value x given its digitized form x_0, in units of ULPs:

$$P_{x_0}(x) = \begin{cases} 0 & if \quad x < x_0 - 1/2 \\ 1 & if \quad x_0 - 1/2 \le x < x_0 + 1/2 \\ 0 & if \quad x_0 + 1/2 \le x \end{cases}$$

Other distributions can be used, given knowledge of the system being represented.

In this approach, a PDF $P_{f(A,B,\ldots)}(x)$ represents probability that some computation f on inputs A, B, ... represents an exact value x. For example, consider the distribution of exact values x that might be represented by adding a constant to some random variable X. The resulting PDF is given by:

$$P_{X+w}(x) = P_X(x-w)$$

In other words, the PDF of the sum is just a shifted form of the PDF of the operand. Likewise, multiplication by a constant w has the form:

$$P_{X\times w}(x) = P_X(x/w) \,/\, abs(w)$$

Addition and subtraction

Consider the arithmetic operation z=y+x, represented as the sum of random variables: $Z=X+Y$. The distribution of Z, assuming independence of X and Y, is

$$P_{X+Y}(z) = \sum_{x,y|z=x+y} P_X(x) P_Y(y)$$

Since $z = x + y$ and $z - y = x$, we can substitute for x:

$$P_{X+Y}(z) = \sum_{y} P_X(z-y) P_Y(y) = P_X(x) * P_Y(y)$$

In other words, the PDF of the sum is the convolution of the two summands' PDFs. The substitution $y = z - x$ could have been made instead, giving a sum over the product $P_X(x)P_Y(z-x)$. This confirms the property of convolution that either assignment of parameters to convolved functions is acceptable. Note, however, that the choice of x or y substitution selects the parameter assignment. That reversibility of parameters derives from the commutativity of addition of PDFs. In later discussion of non-commutative operators, the choice of substitutions becomes important.

There is no basic problem in convolving two piecewise functions. A piecewise function can be written as a sum of non-overlapping segments:

$$P_X(x) = \sum_{i=1}^{N} p_i(x)\,|_{a_{i-1} \le x < a_i}\text{, with } a_0 = -\infty.$$

Convolution of two functions of this form is straightforward:

$$P_{X+Y}(z) = P_X(x) * P_Y(y) = \left(\sum_{i=1}^{N} p_{Xi}(x)\,|_{a_{Xi-1} \le x < a_{Xi}}\right) * \left(\sum_{j=1}^{M} p_{Yj}(y)\,|_{a_{Yj-1} \le y < a_{Yi}}\right)$$

$$= \sum_{i=1}^{N}\left[\left(p_{Xi}(x)\,|_{a_{Xi-1} \le x < a_{Xi}}\right) * \sum_{j=1}^{M} p_{Yj}(y)\,|_{a_{Yj-1} \le y < a_{Yi}}\right]$$

$$= \sum_{i,j}\left(p_{Xi}(x)\,|_{a_{Xi-1} \le x < a_{Xi}}\right) * \left(p_{Yj}(y)\,|_{a_{Yj-1} \le y < a_{Yi}}\right)$$

By linearity, the convolution of sums equals the sum of convolutions of summands. That means that, given piecewise PDFs representing input values, piecewise PDFs represent the distributions of sums.

A constant w can be represented as a random variable with a Dirac delta function $\delta(x-w)$ as its PDF. Convolution properties of the delta function yield the result cited earlier,

$$P_{X+w}(x) = P_X(x) * P_w(y) = P_X(x) * \delta(y - w) = P_X(x - w)$$

Subtraction follows from a combination two rules, multiplication by a constant and addition:

$$P_{X-Y}(z) = P_{X+(-1\times Y)}(z) = P_X(x) * P_Y(-y)$$

Multiplication and division

Multiplication of independent random variables X and Y follows the same general form as addition:

$$P_{X\times Y}(z) = \sum_{x,y|z=x\times y} P_X(x)P_Y(y)$$

Since $z = x \times y$ and $z / y = x$, we can substitute for x:

$$P_{X\times Y}(z) = \sum_{y} P_X(z / y)P_Y(y)$$

The summation can be replaced by integration for continuous PDFs:

$$P_{X\times Y}(z) = \int_{-\infty}^{\infty} P_X(z/y)P_Y(y)dy$$

Note that this summation or integration is almost identical to convolution, except that the subtraction operator in the P_X parameter has been replaced by division. In general, a pseudo-convolution of this form will represent combination of two PDFs according to some binary operation. It is not necessary that the operator be commutative, although non-commutative operators require a little extra care in the variable substitution step.

<Note weirdness when P_X or P_Y has non-zero values in a range straddling zero.>

Similar reasoning gives the PDF resulting from the division `z=x/y` of independent random variables X and Y:

$$P_{X/Y}(z) = \int_{-\infty}^{\infty} P_X(zy)P_Y(y)dy\ .$$

General unary functions

It is usually possible to compute the PDF of a unary arithmetic operation. First, for some random variable X, define the function:

$$F_X(a) = Pr\{X \le a\}$$

i.e. the cumulative distribution function (CDF) of X. Define a new random variable $Y=g(x)$, so that

$$\begin{aligned} F_Y(a) &= Pr\{Y \le a\} \\ &= Pr\{g(X) \le a\} \\ &= Pr\{X \le g^{-1}(a)\} \\ &= F_X(g^{-1}(a)) \end{aligned}$$

Since

$$F_X(a) = \int_{-\infty}^{a} P_X(x)dx\ ,\ \text{it follows that}\ P_Y(x) = \frac{d}{da}F_Y(a) = \frac{d}{da}F_X(g^{-1}(a))\ .$$

This depends, of course, on the existence of $g^{-1}(x)$ and on differentiability of $F_X(g^{-1}(a))$. Given that many $P_X(x)$ have piecewise definitions, piecewise differentiability is usually adequate.

General binary operators

Given some binary operator, $x \bullet y$, it is often possible to define a pseudo-convolution operator $\oplus$ such that, for random variables $Z = X \bullet Y$, $P_{X\bullet Y}(z) = P_X(x) \oplus P_Y(y)$, assuming that X and Y are independent.

First, assume an inverse operator $\circ$ such that $(x \bullet y) \circ y = x$. Then,

$$P_{X\bullet Y}(z) = \sum_{x,y|z=x\bullet y} P_X(x)P_Y(y)\ .$$

Given that $z = x \bullet y$, it follows that $z \circ y = (x \bullet y) \circ y = x$, so we can substitute

$$P_{X \bullet Y}(z) = \sum_{y|} P_X(z \circ y) P_Y(y) \ .$$

If $\bullet$ has only a left inverse such that $x \circ (x \bullet y) = y$, then the pseudo-convolution comes from the substitution $x \circ z = x \circ (x \bullet y) = y$, i.e.

$$P_{X \bullet Y}(z) = \sum_{x|} P_X(x) P_Y(x \circ z)$$

Application

This analysis can be performed automatically, using symbolic algebra to represent and combine the PDFs. Unfortunately, there does not appear to be any simple set of basis functions that includes piecewise constant PDFs and is closed under all of the operations listed above. Analysis requires a fairly flexible system (e.g. *Mathematica*) for handling symbolic algebra. However elaborate the results, though, they can always be approximated by polynomials of limited degree.

Since the PDF of each computed is held symbolically, a number of interesting statistics can be calculated directly:

- The highest and lowest values with a non-zero probability. In a piecewise function, it is trivial to find the highest and lowest values for which the PDF is non-zero. The extrema of the PDF correspond to interval limits in interval analysis, so PDF arithmetic provides at least the expressive power of that technique.
- The mode is a global maximum, i.e. a point at which the PDF's derivative goes to zero. Symbolic differentiation can find $dP_X(x)/dx$ directly. Standard techniques can then solve that derivative for zero. If the mode corresponds to a discontinuity in a piecewise function, the PDF may not be differentiable at the mode. The mode can still be found, in those cases, using other standard maximization techniques. The mode is especially interesting, since the modal value of the distribution is the real-world value with the highest probability of being represented by the result of the calculation.
- The median, i.e. the point at which the cumulative distribution function (CDF) equals ½. The CDF is the integral of the PDF. That means the CDF can be computed symbolically and solved for CDF(x) = ½ using standard techniques. This technique can also locate confidence interval that brackets (for example) 95% or 99.9% of the distribution. Depending on the application, different definitions of the confidence interval may be used [Spr00].
- The mean and variance, $\mu = E[x]$ and $\sigma = E[(x-\mu)^2]$. These can be computed as

$$\mu = \int_{-\infty}^{\infty} x P_X(x) dx \text{ and } \sigma = \int_{-\infty}^{\infty} (x-\mu)^2 P_X(x) dx \ .$$

These expressions can be computed and integrated symbolically, then evaluated.

Since the PDFs are stored symbolically, the designer is free to analyze them in many ways, using thresholds or techniques of the designer's choice. Also, since the basic

computation yields a PDF, the thresholds have clear meanings – they are not just 'magic numbers' a designer picks to achieve some ill-understood result.

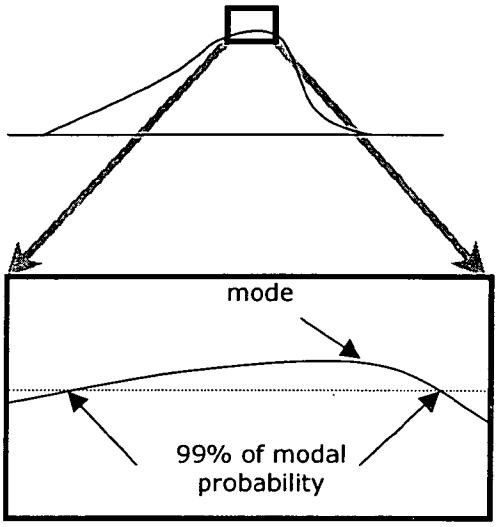
One statistic, in particular, appears to be helpful in deciding how many LSBs are too contaminated to be of any further use. That statistic measures the likelihood (odds) that the computation yields some value exact value y as its result.

Assume a unimodal PDF - a good assumption in ordinary cases. The highest point on the PDF, the mode, can be found readily, and the value of P_X evaluated at that point.

Typical PDFs arising in calculations tend to be flat (but not necessarily differentiable) near the mode. In other words, many values near the mode have very nearly the same likelihood of being the one that this PDF represents. Likelihood is a ratio:

$$P_X(y)/P_X(x) .$$

This may have any value in $[0,\infty)$. Higher values represent higher probabilities that the curve represents y instead of x. A likelihood of 1 indicates that the curve represents equal probabilities that the infinite-precision calculation would have yielded x or y. The ratio is always less than or equal to 1 when x is the mode, but may be close to 1 for many values. Given some threshold, perhaps $P_X(y) \geq 0.99\ P_X(mode)$, the values y in that range are all about equally likely as candidates for the "right" answer, i.e. the value that this PDF represents.



That means that the PDF representing the results of calculation does not clearly distinguish values in that range. For example, given the precision of the result, that range may be 16 ULPs wide. The result contains four bits of precision that are not justified by the accuracy of the calculation, at the chosen level of probability. This information gives a designer good reason to drop four LSBs from the result. Also note that this likelihood confidence interval [Spr00] is somewhat different from a probability confidence interval: it gives the analyst direct control over the probability of confounding results, not just the probability of nearby results.

<TBS: There has to be some kind of Bayesian expression based on $P(x|X_i)$, the probability of an exact result given the computed result, increasing $P(x|X_i)$ by decreasing the number of X_i available. >

Implementation note

The algebraic expressions representing results of computations can become very complex within just a few steps. It may become infeasible to carry the exact form of a result beyond a few additions, multiplications, and divisions. One may, however, approximate a result using polynomials, or a piecewise function using piecewise polynomials.

Happily, piecewise polynomials are closed under convolution. They can create $ln(x)$ terms, however, in the pseudo-convolution that represents multiplication of two random

variables. It should not be difficult to convert the *ln(x)* terms back into polynomials of adequate precision.

As a computation progresses, the degree of the polynomials involved can become very large. It is always possible to approximate the high-degree polynomials using low-degree terms, to within stated accuracy.

Discussion

This PDF analysis consistently makes the assumption that arithmetic operands can be modeled as independent random variables. In fact, that is very rarely the case in real calculations. Consider, for example, the one-dimensional linear regression

$$\beta = \frac{n\sum_i X_i Y_i - \left(\sum_i X_i\right)\left(\sum_i Y_i\right)}{n\sum_i X^2 - \left(\sum_i X_i\right)^2}$$

The ΣXY term is clearly not independent of $(\Sigma X)(\Sigma Y)$, ΣX^2 is not independent of $(\Sigma X)^2$, and the numerator is not independent of the denominator. The exact dependence (joint distribution) could be difficult to characterize, however.

It clear, though, that differences of correlated values tend to be smaller than differences of uncorrelated values. Empirical measurements show this, and suggest that fewer significant bits need to be allocated to differences of correlated terms than to differences of uncorrelated arguments.

Such savings in bit allocation can not be detected using basic PDF-based analysis, since all operands are assumed to be statistically independent.

PDF-based analysis does not preclude other kinds of analyses. In fact, it can accept results from other analyses as inputs. One may still perform the empirical tests on an algorithm, to collect information about bit usage at specific points in a calculation. The empirical frequency results can then be phrased as probabilities (PDFs), perhaps as simple as piecewise constant distributions, and inserted into the calculation stream.

Objections to probabilistic analysis

PDF arithmetic avoids many of Kahan's criticisms [Kah96] of theoretical probabilistic error analyses:

1. That they do not provide adequate information about the extreme cases of error accumulation. PDF arithmetic is a proper superset of interval arithmetic – the minimum and maximum values with non-zero probability are the same as the bounds used in interval arithmetic.
2. That the make inappropriate use of the Central Limit theorem. PDF arithmetic does not rely on the Central Limit Theorem, though not for Kahan's reasons. PDF arithmetic does not rely on distributions (e.g. Gaussian or χ^2) chosen *a priori* as representative of errors. Approximations, when made, are meant to preserve the PDFs already found, but in terms of simpler expressions. They do not impose a preconceived form on the PDFs.

3. That they assume all operands contribute error terms of different magnitudes. PDF arithmetic automatically acknowledges the magnitude of each error term – if a few expressions dominate the computation error, they also dominate the PDF of the computation's result..

Of course, PDF arithmetic avoids the problems inherent in sampling, as well. PDF arithmetic, as proposed, makes the assumption that operands to binary operators are independent – an assumption known to be false in many computations. Short of completely symbolic evaluation, however, it is not clear how to address this problem.

Bibliography

Cov91 Cover, Thomas M. and Joy A. Thomas. *Elements of Information Theory*. John Wiley and Sons. 1991

Kah96 Kahan, W. *The Improbability of Probabilistic Error Analyses for Numerical Computations*. Copied from http://www.cs.berkeley.edu/~wkahan/improber.pdf on 1 Jul 2003.

Spr00 Sprott, D. A. *Statistical Inference in Science*. Spring-Verlag. 2000.

Zwi03 Zwillinger, Daniel. *CRC Standard Mathematical Tables and Formulae*. Chapman and Hall / CRC. 2003

Martin --

I came away very satisifed with our meeting with Vajda. It really sounds as if our initial docking chat with Zhiping represented (more or less) state of the art. Here are the things that sound especially good to me. For current discussion, I'll talk about problems being O(N) if the total number of voxels is N.

1) Scoring is only slightly worse than I thought. Sandor's first pass uses simple scoring (miss/intersect/collide), the second pass uses isotropic forces (e.g. point charges), later passes may use anisotropic forces (e.g. electric dipoles, aromatic aligment). The first two are still interesting enough to be worthwhile, if only as the pre-filter into the late (expensive) passes, and also amenable to simple hardware.

1a) Saturated small integer arithmetic looks good for scoring. I'll accept a few collisions representing places where flexible molecules would have flexed, but too many (saturation) would be bad. I'll count up surface/surface interactions, but too many (saturation) automatically trigger more expensive scoring (unless collisions veto the configuration).

2) I am reasonably sure that correlation can be done O(N), if I can load one whole molecule grid into the chip first. It puts massive load (1:N) on many signal lines, but I'll be happy to turn the clock down. I won't stand by this claim until I put some pieces together, but I'm very optimistic. The actual cycle would be: a) load O(N) voxels for the substrate, b) pump in O(N) voxels for the ligand, c) read out O(N) voxels of correlation. Because of fanout, b) might be slower than a,c). The big Xilinx chip can probably handle a 32**3 grid, which may be enough for a low-res filtering pass. If grid has to be cut into C pieces so that each piece fits in the chip, the time should be O(NC)

3) I am reasonably sure that pitch/yaw/roll (PYR) rotation of the molecule grid is not needed, if I can rotate the axes and step along the rotated axes. In other words, the rotation problem becomes O(k) additional work per PYR choice, where k is small. This will increase the amount of work in step 2, but not by more than sqrt(2)**3 [maximally misaligned rotation].

All this adds up to O(CNR) data movements, where R is the number of different PYR rotations evaluated, and C is the number of blocks that must be used to represent the whole grid. This isn't the fastest imaginable algorithm: it touches all voxels, including the "uninteresting" interior and "vacuous" exterior. I also have some ideas based on low-resolution prefiltering that could help, but I need to develoip those thoughts.

-- tvc

Hardware acceleration for computed molecule interactions

One class of problems in computational biochemistry deals with interactions between molecules. Two forms of the problem are known as "protein docking", in which two large molecules interact, and "drug screening", in which small molecules (ligands) are tested for interaction with a large molecule (the substrate). Many computational approaches are currently in use for evaluating such interactions. It is common, though, for the computation to run in two or more phases: relatively quick scans for molecule translations and rotations (jointly referred to as a positioning) that seem likely to interact, followed by detailed evaluation or refinement of the initial juxtaposition.

Initial scans often use voxel models of the ligand and substrate. Different voxel values represent the exteriors, interiors, and surfaces of the molecules, and may represent chemical features such as hydrophobicity. The modeled substrate is typically held fixed while the modeled ligand is rotated and translated into many candidate positions around it. A score is computed for each positioning, based on the sum of interactions between coincident voxels. Collisions between molecule interiors contribute negative scores, and favorable surface interactions create positive scores. Once a score is computed for every possible position of ligand with respect to substrate, positions with scores above some threshold are collected for further processing.

A basic scoring algorithm just computes the correlation of the 3D grids. One variant uses complex numbers to represent voxels: 0 for exterior, positive imaginary for interior, and positive real for surface voxels. The sum of voxel products adds in negative values for imaginary-imaginary interactions representing interior collisions. Correlation is essentially convolution of the two voxel grids, which is computationally intensive. The standard approach is to compute the 3D Fourier transform (3DFT) of each molecule, multiply the transforms element-wise, and inverse-transform the result. This must, of course, be repeated for ever three-axis rotation of the ligand. The 3DFT and inverse involve $O(n^3 \log n)$ computations, but that is a significant improvement over $O(n^6)$ computations for a naïve convolution. Still, rotation and transformation of voxel grids much larger than 128^3 can be time-consuming.

The remainder of this paper describes a hardware accelerator for the rotation and convolution.

Scoring

The first step in hardware acceleration is to review the purpose of the calculation. Here, each positioning of ligand and substrate is evaluated for two factors: presence of desirable surface interactions and absence of interior collisions. Small numbers of collisions may be acceptable, perhaps due to roundoff error in quantizing the molecules. This rigid model of a flexible molecule also has inherent imprecision, so minor or shallow collisions may not be significant. Beyond some threshold, though, the number and severity of colliding interior voxels may become unacceptable. The computation may proceed even after that threshold is reached, but additional collisions don't make relative positioning any worse. In other words, a saturating counter or accumulator may represent the collision detection logic adequately.

Once the positioning is known to lack unacceptable interior collisions, its surface to surface interactions should be evaluated. These are also detected by coincidence of surface voxels in the substrate and ligand. Larger interaction areas are generally better, and interaction areas above some threshold may automatically qualify the positioning for further evaluation. Scores above the selection threshold don't matter, so a saturating accumulator can model interaction area scores, also. Other scoring functions, perhaps based on hydrophobicity, local charge, etc, can be computed in similar ways.

The net result of this reasoning is that the score for any positioning of ligand with respect to substrate can be summarized in two bits: it does or does not exceed the collision threshold, and does or does not exceed the area threshold. This result can be extracted easily from collision and area accumulators with modest numbers of bits.

Convolution

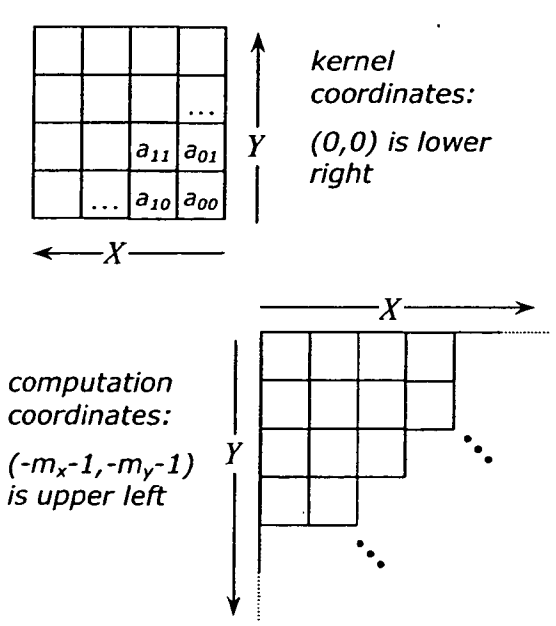
The reason for the 3DFT and inverse is that $0(n^6)$ computations become prohibitively expensive for even modest increases in n. Although costly itself, the 3DFT is still cheaper than a naïve convolution.

When considering an implementation in custom, highly parallel hardware, it is more meaningful to consider complexity as a product of time and hardware units. The $O(n^6)$ behavior is unavoidable in naïve convolution. The convolution can be phrased as $O(n_T^3\ n_{HW}^3)$ however, where n_T represents the time and n_{HW} represents the hardware required for the operation. Assuming unlimited hardware, for the moment, this reduces time complexity of the convolution to $O(n^3)$, which is less than the polynomial complexity of the 3DFT. The remainder of this section describes a hardware implementation that meets this description.

Consider a convolution of some kernel (representing the ligand) over some data (representing the substrate). For current discussion, it is not necessary that the convolution kernel overlap the data completely. Any part of the kernel that passes the edge of the data is ignored. Those elements of the kernel contribute 0 to the result.

Represent the computation grid as a rectangle n_x+m_x-1 plus n_y+m_y-1. (This two-dimensional discussion will be seen to generalize readily to three dimensions.) Number the grid coordinates from $-m-1$ to $n-1$ along both axes. The coordinate of the computation grid will represent a position of the kernel with respect to the data; that cell of the computation grid will accumulate the sum of kernel×data products for that kernel position. It may help to think of the data grid as aligned to the (0,0) position within the computation grid.

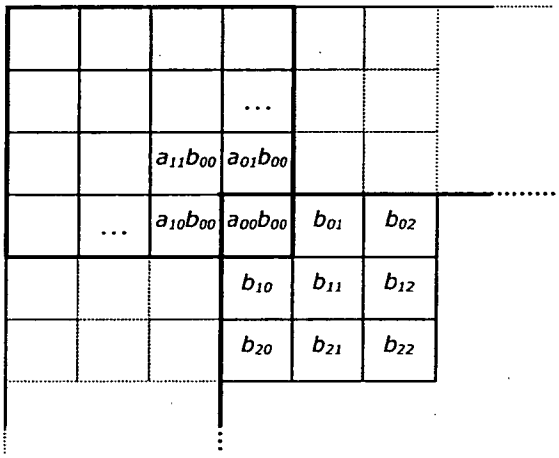
Flip the usual indexing of the kernel so that its low indices are at lower right and high indices are at upper left. Figure 1, at right, illustrates these indexing conventions.



Position the kernel at upper left in the computation grid and observe the following:

- the lower right corner of the kernel starts at grid position (0,0), corresponding to data position b_{00},
- the kernel covers precisely the area of the computation grid in which convolution sums involve b_{00}, and
- the kernel value a_{ij} at each computation grid cell is the coefficient of b_{00} in that cell's convolution sum.

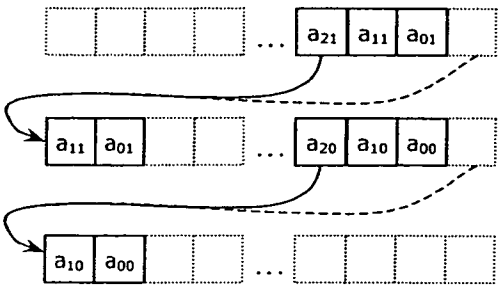
In other words, one may broadcast the b_{00} data value to the kernel overlaid on the computation grid, and increment each convolution sum by the $a_{ij}b_{00}$ at each computation cell. All of the multiply-accumulate operations are carried out in parallel, using hardware dedicated to each computation cell.



Next, shift the kernel right by one position, so its lower-right corner corresponds to data element b_{01}. Again, each kernel element a_{ij} overlaps the computation grid cell at which the convolution sum includes the $a_{ij}b_{01}$ term. Add that product to the convolution sum at each computation cell. Continue stepping the convolution kernel across the computation grid until the rightmost columns of the kernel and computation grid align. The logical operation to perform next is to reposition the kernel at the left edge of the computation grid, one row down, step it across that row, and repeat row by row.

It can easily be seen that this computation requires as many cycles as there are elements in the substrate voxel grid. The data-serial communication paths can then be modified slightly to link all computation grid cells into one shifter, and to fetch results at the rate of one result per cycle.

So far, this computation involves two communication paths: a broadcast of the b_{ij} value to all kernel elements and a shift of kernel elements across the computation grid. It would require significant additional communication paths shift the whole kernel down one and back to the start of the line. The same net effect can be reached with only one additional communication line per row, however. That line appears as the solid arrow in the diagram at right.



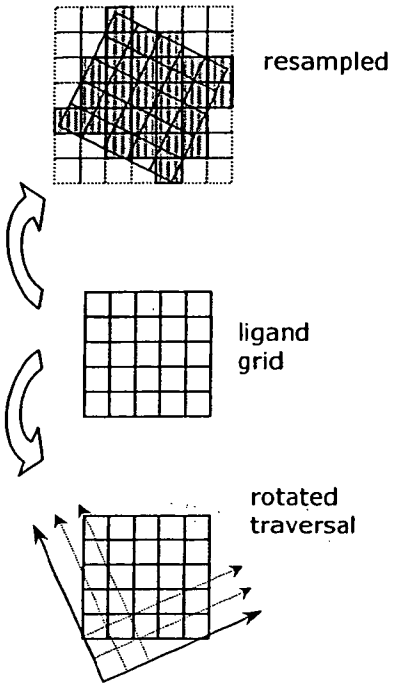
As the kernel values shift across the computation grid, they can also be shifted into the next line. After the original kernel touches the right-hand side of the computation grid, it is no longer useful. At that point, however, it is properly positioned on the next line, ready for the next set of computations. A small amount of book-keeping is required to disable computations on the wraparound values until the entire kernel is ready, and to

stop using stale values at the end of each row in the computation grid. Although this scheme is illustrated using a two-dimensional computation grid, it extends readily to three dimensions.

Once the convolution kernel has reached the end of the computation grid, convolution sums are stored in each cell in the grid. At that point, computation grid is reconfigured to include all cells in the shifter chain (the dashed line in Figure X). The convolution results are loaded broadside into the shifter, and the host unloads the results serially.

Rotation

The remaining problem is to rotate the ligand before performing correlation with the substrate. The traditional approach is to create a new voxel grid, large enough to handle the rotated ligand, and to resample the ligand voxel model into the new grid. This requires additional buffer space, possibly as much as 520% the size of the original buffer.



The current implementation skips rotation as a specific step by rotating the axes along which the ligand array is traversed. The traversal axes are phrased as some origin point (x_0, y_0, z_0), plus three direction cosines for each axis, (x_x, y_x, z_x), (x_y, y_y, z_y), (x_z, y_z, z_z). These can be represented as signed, fixed-point numbers, with just enough precision to avoid rounding errors over the maximum extent of the rotated ligand. Each increment along a traversal axis has the following form:

$$\begin{array}{lll} W_x = x_0 \text{ initially}, & \text{then} & W_x = W_x + x_w \\ W_y = y_0 \text{ initially} & & W_y = W_y + y_w \\ W_z = z_0 \text{ initially} & & W_z = W_z + z_w \end{array}$$

where W represents one of the traversal axes X, Y, or Z. Converting traversal axes to ligand axes (X', Y', Z') is straightforward:

$$\begin{aligned} X' &= \lfloor X_x + Y_x + Z_x \rfloor \\ Y' &= \lfloor X_y + Y_y + Z_y \rfloor \\ Z' &= \lfloor X_z + Y_z + Z_x \rfloor \end{aligned}$$

where $\lfloor s \rfloor$ is the greatest integer $\leq s$.

The *floor* function actually implements true rounding to the nearest integer, given an appropriate bias in the initial (x_0, y_0, z_0) values. Before using the (X', Y', Z') values as ligand grid indices, each index is checked to make sure it lies within the ligand grid's range. If any index is out of range, the ligand voxel value is taken to be 0, i.e. exterior to the molecule. The small effort in computing the 12 access parameters is performed by the host. The host loads new access parameters at the start of each new convolution. Note that the ligand voxel values need to be loaded only once, the reused for each set of access axes.

%%
%%

??? group notes by Rick Barton, Martin Herbordt, Tom Van Court

%%
%%

Goals/Motivation:

- -- Make bioinformatics applications go fast -- a factor of 100 faster than running on a PC is the goal.
  - --> Go fast through special purpose hardware support that is transparent to the end user. An accelerator that plugs into the PC as easily as adding a memory card.
    - --> Hardware support through language/function-library infrastructure to make the plug-in card as usable to the application developer as a standard high-level programming language.
- -- Classic Architecture/Systems Argument --> Combination of emerging application and viable technology has created a potential "sweet spot." The major problem in filling this sweet spot is usabality.

Scenario

- -- Usability requires that the end users be given the tools so that they can, with minimal training, use them.
- -- Who currenty uses bioinformatics applications? Some biologists use them to supplement "web lab" experiments. However, perhaps the primary users are the bioinformatics support staff that work with the biologists.
- -- Who creates bioinformatics applications? The main-stream is now common domain. Many are commercially available. New ones are created by bioinformatics support staff.
- -- Bioinformatics support staff are skilled programmers, but cannot be expected to have special system/hardware skills.

End Applications:

- -- Genomics
- -- Proteomics, etc.

Mid-Point Applications:

- -- Partical string matching
- -- Bayesian Inference
- -- Search
- -- Linear Algebra and statistical techniques

Who are the customers?

- -- Bioinformatics practitioners through the computational support staff in bioinformatics groups.

Enabling Technologies (general)

- -- VLSI process curve: especially for gate-arrays and FPGAs
- -- PCI slots/PCI boards
- -- FPGA-based boards and development systems
- -- IP-based ASIC
- -- Compilers for translating high-level language programs to hardware

Competition:

- -- Supercomputers/MPPs
- -- Linux clusters

-- Special purpose boxes (Sun, Celera)

Why we are better
-- Cost in terms of initial hardware cost and system maintenance.
-- Performance in terms of throughput (number of experiments), and response time (for data exploration).

%%
%%
Particular to CAAD Lab (us -- we need to come up with a project name!)

Multi-pronged approach:
1. FPGA
   a. Basic hardware design, exploration of technology/design options.
   b. Language/compiler
   c. Function libraries
   d. infrastructure for test, proof-of-concept
2. Other technologies -- use of gate arrays, standard cell, SIMD processing, or other technologies. Benefit is if we find programmable hardwired components. Or if we need stuff that just does not work in FPGAs.

What we need to demonstrate:
-- we can create hardware that is much better than a PC
-- we can do this for lots of bioinformatics applications
-- we can create infrastructure so that other people can do this to, cheaply, effectively, etc.

Candidate applications:
-- "find the set of genes that ..." using linear algebra
-- bayesian nets -- optimization through search of bayesian nets.
-- partial string matching

Why we think this is a good idea, or why the technology and application are coming up a match:
-- data set size and reuse (working set size)
   * only processing 30-100 thousand genes ???
-- ratio of computation to data
-- data types
   * often processing 2-3 bit quantities
   * rarely need lots of precision ???
-- key algorithmic components look great in hardware:
   * associative processing: search, matching, leader election, reduction
   * systolic processing
   * configurable cellular automata (processing elements)

What we are not doing:
-- large-scale database stuff.
-- gene sequencing (although we would probably be good at that)

%%
%%
Characteristics of problems we are likely to be good at:

Data sets that (barely) fit on a chip, at least for local computation. Say a 1-100 million bits at a time.

Simple, highly repetitive evaluation functions

Variety of algorithms

Variety of data types, but with little precision needed through most of computation

Multiplication of small numbers is great, division terrible.

Complexity at least quadratic and hopefully cubic in the size of the data.

Use associative algorithms. Very highly parallel: search/process with respect to sample data, distribute results, order results (max,min, etc.), combine results.

%%
%%
Characteristics of Biocomputing Problems:

[See Rick's writeup: "A Canonical Form for Biocomputing Problems"]

Search problems with the following characteristics:

1. Large parameter set: large number of dimensions to be searched, compressed, etc.

2. Simple Performance Criteria -- score function of parameters

3. Decomposable search strategy or score function

4. Large but manageable sample data set

%%
%%
Previous work: [TVC edited by MCH]

What's commercially available:

-- Celoxica: general C to FPGA compilation
-- Sun cooperation with ??? for large servers
-- Celera offshoots
-- Paracel: BLAST accelerator, gene matcher
-- Accelchip.com: Matlab to verilog/vhdl/DSP cores ASICs

Pevious academic work in HLL to FPGA
Para, BRASS/Garp, RAW, Cameron, PipeRench, chimera, etc.

%%
%%
HLL to Gates (random thoughts): [TVC edited by MCH]

[MCH: We need to review state of the art of the Verilog/VHDL world of commercial use. This is likely to be the "true" state of the art -- HLL to gates is the holy grail of the EDA business and as always, need to follow the money (and Cadence and Synopsys have a lot of it) and leverage its results!]

Cameron: work for heavily stereotyped inner loops
Single assignment languages seem most common
Some work with UNITY language -- temporal logic language aimed at program proving (why not Z notation?)
Other languages: Occam, DIL (dataflow)
Mixed output: C and gates for specific processor + FPGA structure or C-only for simulation.
Range of solution types: (near) full C language to 100% FPGA is very extreme. Processor/gate tradeoffs, limited language (e.g. Cameron, SA-C) may have canned library of gate array (GA) blocks, canned GA blocks may be parameterized to some extent (like traditional special-purpose function logic function generators), reconfigurable processors.
Need to characterize problems to be addressed, to biuld programming constructs tailored to target problems.

%%%
%%%
FPGA Issues (random thoughts): [TVC edited by MCH]

Load/unload times may dominate: 10**2 to 10**3 pins, but millions of data values. Recomputing results may be easier than unloading and reloading. Massive reuse of data avoids load/unload costs.

Special GA features will affect design: memory/FIFO (kind and amount), dedicated gates (e.g. multiply, carry??), processors.

Latest FPGAs (as always) are pretty incredible -- they not only have many more gates, they have much more specialized logic that one must try to take advantage of (e.g. a number of processors).

It's not clear that academic compile to silicon strategies have caught up with commercial technology.

Coupling to host system.

Why FPGAs look good:
- -- Algorithms may not have settled yet -- FPGAs don't commit to algorithms
- -- Algorithms are amenable to highly parallel execution of small simple algorithms (e.g. 1000 string matchers on a chip, feed sequences past them, cooperate on sharing results)
- -- Out lines of some algorithms (e.g. some steps in BLAST) seem well standardized, small operations in inner loops differ (see Cameron)
- -- Revive older ideas with new technology: associative processing, systolic arrays
- -- data is variable sized and often very small, but may get bigger
- -- computations have multiple parts that have qualitative differences, so would lend themselves to reconfiguration.

%%%
%%%
Case Study 1: (need a name)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Statement:

Some thousands of experiments correlating activity as measured by microarrays of some thousands of genes with a binary outcome.

See which very small subset of genes (say 3) correlates the best with the outcomes.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Our method:

Given easily constructed or obtained hardware (PCI card, FPGA, memory, PC), how do we program that hardware to do this really fast. A parallel investigation is to examine less easily constructed hardware, although this should be a last resort.

For all combinations of 3 genes out of some thousands with some thousands of experiments, want to find combination that lends itself best to being a discriminator. Looking at each combination using standard method for solving overdetermined systems:

(Notation: AT <---> A transpose A-1 <---> A inverse)

Distance best solution x is from being a perfect discriminator =
(AT b)T (AT A)-1 (AT b)

b is 1000 x 1 -- each element is 1 bit
A is 1000 x 4 -- each element is 2-4 bits

To do AT x A:
- -- AT x A is 4x1000 X 1000x4 we do 16 long dot products
- -- For each dot product, multiply pair of 2-4 bit elements to get 4-8 bits and accumulate.
- -- For each dot product, accumulate 1000 4-8 bit elements to get a 14-18 bit quantity.

To do (AT x A)-1:
- -- To invert a 4 x 4 matrix of 14-18 bit elements. If we use determinants, then we need only one division per element of the inverse. Hopefully, we could avoid doing this division at all by saving numerator and denominator through the rest of the computations.
- -- Determinant is sum of 24 terms, each the product of 4 terms. This leads to each term being from 56 to 72 bits. Actual determinant is probably going to be much smaller since half the terms are subtracted.
- -- If we use Gauss-Jordan, then intermediate terms stay small. Does it help that matrix is symmetric? However, there is no way to avoid division.
- -- How do do an efficient division?

To do AT X b:
- -- 4x1000 X 1000x1 and get 4x1
- -- each dot product is the sum of 1000 2 to 4 bit times 1 bit elements or 13-15 bits in all.

To do (AT b)T (AT A)-1 (AT b)
- -- 1x4 X 4x4 X 4x1 to get a single number
- -- sum of 4 13-15 bits X ??? --> 15-17 bits + ??? bits
- -- sum of 4 13-15 bits X (15-17 + ??? bits) = 30-24 + ??? bits

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Algorithmic complexity

Algorithmic complexity (simplest form)
1. Combinations of number of genes by number selected for significance
   e.g. 1000 3 at a time or 166 million
   2. number of dot products = number of genes + 1 squared, e.g. 16
      3. length of vector = number of experiments, e.g. 1000
   4. inverse is order 16 cubed operations or 4K

other stuff is smaller
166000000 x (16 x 1000 x 2 + 4000) = 166 x 36 x 1000000000 operations
= 6 trillion operations or roughly 3 hours

Algorithmic complexity (reuse dot products)
still need to do 166 million inverses, but could avoid most of the repeated dot-product computations: only 1000x1000 possible dot procucts which gets you about 10**9 operations which is very fast.

Implications of algorithmic complexity results: seems to point to need for fast inversion (!) for problems of this size.

HOWEVER: the data sizes make the complexity go up very quickly.
- -- number of genes is in the exponent (combinations)
- -- number of genes gets raised to the number of genes "looked at" which we can assume is at least cubic.
- -- number of experiments seems to be a linear term since it only affects the lengths of the vectors.
- * If we are doing 10K experiments and 10K genes, but still looking at 3 genes, then we have 166 BILLION combinations, and each dot product takes 10K ops rather than 1K.
- * If we are doing 1K experiments and 1K genes but looking a 4 genes, then we have 42 billion combinations. Also, 25 dot products and a larger inversion which slowly starts to dominate as gene set size increases.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Our implementation

Note on division: if we can get it so that we only do one division per combination of 3 genes, then we can leave the 166 million combinations as ordered pairs and stream them off to the PC or other coprocessor to finish.

Fast computation:
- -- dot products are done by streaming vectors through multiply accumulate units.

Parallelism and reuse: Lots of levels of this.
- -- We want to do as few computations as possible (certainly). If we can really save a tremendous amount here, then we also need to apply this to serial and MPP solutions.
- -- We can't keep everything hanging around, so reuse has three possibilities:
   1. use many times in succession so compute once and keep on chip.
   2. use many times, but distributed.
      a. If much work to compute, then compute once and keep off chip.
      b. If easy to compute, then compute each time.
- -- Reuse has many levels.
   1. Complete dot products get used 166 times apiece.
      1000x1000 dot products. Could compute each of the 1000000 dot products at once, offloading as they are computed, then reload to do inverses.

2. Or, (still for dot-product) could do some subset, say all combinations with the first 10 genes, etc.
3. For inversion, computing determinant terms -- perhaps could reuse many of the terms across determinants. But perhaps not since they are permutations with one entry per column and each 4x4 matrix differs from each other 4x4 matrix in at least one row.

-- Computing in parallel can be done many ways as well

1. depending on reuse more or less computing may be called for. Would create a gigantic dot-product computer where we stream on the the 1000 by 1000 vectors and compute all 1000000 dot products simultaneously. Actually a subset depending on # of I/O pins, computing area, pin bandwidth, etc.
2. To keep results from dot-product on chip as long as possible, may be good to do inversion before off-loading. Can partially reconfigure, or pipeline dot-product computation with inversion.

Hardware small matrix inverter -- the advantage we have in hardware is that we can hardwire things like finding pivots and swapping rows. Problem is that division is very difficult.

Hardware matrix solver -- similar to above

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Sensitivity to real data

Can preprocessing get rid of most candidates?
Can we filter single genes for correlation first and eliminate many?
Where do we get data?
Do genes fall into "equivalence classes"?

At a numerical level, the "back-of-the-envelope" data size calculations above resulted in data sizes that are much larger than are likely. But we won't know how much larger without seeing real data.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
To test our approach:

In General:
Compare our method with standard methods. Idea is that we can do something much much faster using hardware acceleration. This is the first proof of concept. Must then show that we can make such acceleration usable for other people.

More Specific:
Two areas -- standard single researcher tool (PC/workstation) and standard parallel processor (MPP/Cluster).
Packages available -- many linear algebra packages are available for all common platforms: Numerical Recipes, LAPACK, LINPACK, BLAS, etc.

Methods: SVD-based? Still best considering possible reuse? Should probably do direct implementation of what we do in the FPGA, as well as other implementation. (different types of speed-up are interesting)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Other guides to what we should work on:

Concentrate on techniques which will be universally (or at least most likely) applicable and so that will be most likely to be reused in other applications. These include basic linear algebra functions, as well as any functions that use the canonical set of associative functions.

%%
%%
Case Study 2: Miscellaneous search approaches (various contributions)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[From Tom's blurb]

Machine learning: deduce boolean formula that describes some boolean data set typically only 100 data values, 1-bit values, does not tolerate any mismatches

Bayesian Inference Networks: information theoretic metrics, hidden Markov models, are they applicable? Do they fit FPGA model?

Tree searches: exact match versus nearest neighbor. Searching can be split across processors.

Data Mining

Clustering Algorithms

Highly parallel search:
If a tree or search space is computed dynamically, then the tree can expanded out to the size of the processor array and then distributed. After that, each subtree is expanded locally. Use global data collection and broadcast to keep track of stopping and pruning criteria. Reallocate subgraphs as necessary. This is certainly good if the search space is dense. Also good if not too much data is needed to do local evaluation. E.g. if every node only needs a few hundred bytes of data rather than a megabyte.

%%
Bayesian network models of genetic regulatory networks based on microarray data

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Biology of this -- method of understanding a genetic regulatory network.

For some process (e.g. response to some stimulation?) cells take some action. This action is determined by a complex series (network) of chemical reactions. This reaction sequence is regulated by a similarly complex sequence of genetic actions. As these genes are "turned on" they can be detected (indirectly through the protiens they produce?) through the process of microarray analysis. However, since the time and chemical scales of the each microarray assay(?) is orders of magnitude larger than each step in the chemical process, biologists can't just do analysis at various predetermined timesteps. Therefore, each assay is a dirty snapshot of various events taken at random times of the process. However one can assume that the events recorded have some correlation, e.g. some are finishing, others beginning, but these are likely to have some relationship to each other. Problem is to find the causal connections among gene expressions. Bayesian network is one method of finding the causal connection that best matches the data.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[See Rick's blurb]

Definition -- Bayesian network consists of
-- acyclic digraph G in which nodes represent expression levels of different genes and edges indicate a direct dependence of expression level of one gene on another.
-- Set of probability distributions theta consisting of a) marginal probabilities for the nodes in the digraph with no parents and b) conditional probabilities for the nodes in the digraphs with parents.

Given G and theta, the joint probatility of the observed expression levels for the nodes in G can be calculated under the additional Markov assumption that the expression level of any node in G is independent of nodes which are not descended from it. Under certain simplifying assumptions regarding the prior probability distribution of the pair (G,theta), a computationally simple formula can be derived for the posterior probability of G given a set of observed expression levels.

-- Parameter set = all possible digraphs G
-- Scoring function = posterior probability of G
-- Sample data is the set D of observed expression levels

Computational Methods:
-- scoring function can be computed in parallel with each each processor mapped to a node in G.
-- Use a local serach technique such a greedy hill climbing with a large number of indepedent realizations of G. Local changes to G consist of operations such as edge addition/deletion/reversal.
-- Another local search-based method: Each node is given a copy of G through broadcast. Each node varies G in a different way (looks in a different direction) in parallel. Best result is computed through leader election. New graph is again distributed to array and process repeated.
-- Another method of parallelism: try differen parts of the space as seeds in parallel.

%%
%%
Case Study 3: DNA and other matching (various contributions)

Find similar/identical stretches of DMA/amino acids

Large numbers of samples to test against one another

Matching rules: exact, insertions/deletions, variable weights (PAM-150, Dayhoff matrices) for different kinds of mismatch

See BLAST -- lots of heuristics applied to reduce search space, lots of combined techniques.

May be ameneble to CAM (associative processor) or other finite automaton string matcher.

Associative algorithms often seem to have two parts: matching algorithm along the length of each data word (e.g. Hamming distance), then one acrss all

words (e.g. find smallest among the Hamming distances). May represent a generic framework to be filled with various functions.

%%
%%

Various Things to do: (random thoughts)

Set up design infrastructure
 -- Foundation
 -- Cadence

Learn to use design stuff
 -- Verilog language and simulator

Learn existing FPGA technology and how to take advantage of it.

Set up lab so that we can do remote processing onto Hagrid and successors.

Get serial and parallel versions of case study 1 working.

Find more applications.

%%
%%

Array Control for High Performance SIMD Systems[‡]

Martin C. Herbordt
Department of Electrical and Computer Engineering
336 Photonics Center; 8 Saint Mary's Street
Boston University, Boston, MA 02215
email: herbordt@bu.edu; phone: (617) 353-9850

Jade Cravy
GDA Technologies
2071 Junction Ave.; San Jose, CA 95131

Honghai Zhang
Department of Electrical and Computer Engineering
University of Iowa
Iowa City, Iowa 52242

*This work was supported in part by the National Science Foundation through CAREER award #9702483, by the Texas Advanced Research Program (Advanced Technology Program) under grant #003652-952, and by a grant from the Compaq Computer Corporation.

[‡]A preliminary version of this work appeared in the Proceedings of the 5th IEEE International Workshop on Computer Architecture for Machine Perception Workshop (CAMP 2000).

Abstract: Although arrays of SIMD PEs can be built with very high operating frequencies, problems exist in keeping the array busy. The inherent mismatch between host and array makes it difficult to maintain high array utilization: either the rate of instruction issue is very low or PE data locality is compromised, having the same effect. Our solution is based on an array control unit (ACU) design that expands macroinstructions in two stages, first by data tile and then into microinstructions. The expansion itself solves the issue problem; decoupling the expansion modalities maintains data locality. Several issues involving host/ACU interaction need to be resolved to effect this solution. We present experimental results showing that our approach delivers substantial improvement in memory hierarchy performance: a cache of only one fourth the size is sufficient to achieve the same performance as previous approaches. We also describe our implementations which demonstrate that achieving gigaherz instruction issue with current technologies is plausible.

1 Introduction

SIMD arrays retain their niche in vision (see CAMP '97 [34] and CAMP '00 [8], e.g. [2, 5, 9, 10, 13, 15, 16, 27, 29]) and graphics (e.g. the PixelFusion graphics engine [31]). Especially promising are the prospects of SIMD arrays in systems-on-a-chip, particularly when based on SRAM cores [30] or fused with CMOS sensor arrays [26]. Other uses include network processors [11], signal and image processing [33], and bioinformatics [19].

One reason is that SIMD arrays can be built with large numbers of processing elements (PEs) on a single chip to deliver tremendous per-cycle processing capability. Another reason is that it is straightforward to build PEs with very high operating frequencies: except for clock and instruction issue, all of the paths can be made very short. And clock distribution and instruction issue can be optimized with the same methods that are used to distribute analogous signals in high-performance microprocessors (see, e.g. Bolotski [4] and Herbordt et al. [23]).

The basic problem in constructing SIMD array-based systems is that, while PEs can be built to be very fast and while instruction issue and clock can keep up with the PEs, *determining* which instruction to issue next can be significantly more difficult. This is a consequence of the asymmetric, multithreaded nature of SIMD program execution: a host executes the serial (main) thread, including performing the scalar operations and determining program flow, while the array of PEs executes the parallel thread as determined by the host. Since the PEs have been relieved of the control and bookkeeping responsibilities, it is often the case that ten or more serial instructions are executed for every parallel instruction. This is one of the fundamental advantages of SIMD arrays. It also leads to a serious problem: Even with the high-speeds of modern microprocessors, this scenario could easily leave a PE array idle most of the time.

One improvement that has been used previously (e.g. by Gealow [18]) is *instruction expansion*. Balance between host and array is achieved not by having the host send every PE instruction to the array, but rather, only having the host send *macroinstructions*. These macroinstructions are then expanded—by the array control unit (ACU) on the PE chip—to from several to several hundred PE instructions. The expansions themselves can be either

stored in a microstore on the PE array chip, expanded on-the-fly, or a combination of the two.

This simple instruction expansion scheme, however, usually has a major negative consequence: *Much of the locality within the PE array's data stream is lost.* To explain how this occurs, we first describe how instruction expansion is possible in the first place. There are two factors, standard microcoding and tiling. The first is analogous to a microcoded uniprocessor: if a 32 bit add instruction is executed on a PE with an 8 bit ALU, then it must be expanded to at least 4 PE instructions. The second results from a mismatch between number of elements in a parallel variable (e.g. the number of pixels in an image) and the actual number of PEs in the array. This mismatch is often dealt with by mapping each element (e.g. pixel) to a *virtual* PE (or VPE). Each physical PE then emulates the appropriate number of VPEs [25]. Note that fine-grained arrays have more expansion due to the first factor and less due to the second than coarse-grained systems. In either case, expansions of a 1000-1 or more are common for current practical systems. Of this, at least 64-1 is due to tiling.

The locality problem follows from the need to emulate VPEs. Executing each macroinstruction in its entirety before continuing to the next macroinstruction (i.e., through all tiles) can multiply the size of the working set by up to the tiling factor. Since the working set is likely to be large even without VPE emulation (in bytes if not in number of images), the consequence is that PE data is unlikely to remain in the higher levels of PE memory (e.g. registers, on-chip SRAM, or cache) long enough to take advantage of the previous access. This effectively limits the latency of each PE instruction to the memory access time. Although such a bottleneck is undesirable in any system with a memory hierarchy, it is especially troublesome for an inherently pin-limited system such as a massively parallel array-on-a-chip.

In systems where the host generates and transfers all PE instructions directly to the PE array–that is, with no expansion–the loss-of-locality problem is avoided by executing PE instructions "within each tile" for as long as possible. Generally, this is until either inter-tile communication is necessary or a reduction hazard is reached; only then does execution begin of the instructions that operate on data in the next tile. This approach maximizes locality of

data references, but sacrifices macroinstruction expansion on the PE chip.

What we describe in this article is an on-chip array control unit (ACU) that reconciles the apparently incompatible goals of enabling instruction issue at the PE operating frequency and maximizing PE data reference locality. The method is to preload entire "basic blocks" of macrocode onto the PE chip and then let the array control unit (ACU) handle the expansion. The ACU sends PE instructions of the entire basic block for a single tile before doing the same for each succeeding tile. Locality is preserved because the execution thread remains within each tile for a significant time before that tile is "rolled out" and the next tile "rolled in."

In order to make on-chip basic block expansion work, various issues need to be addressed; their solutions–together with the experimental results showing the benefit of this approach–provide a primary contribution of this work. One issue is dealing with the dynamic address computation required by on-the-fly expansion. Another is dealing with scalar variables which are needed by some PE instructions. The values of these scalar variables are often not known until they are computed by the host *after* the basic block has been sent to the ACU and just before the scalar is needed by the array.

The overall significance of this work is that it makes viable larger SIMD arrays that can work on larger problems than would otherwise be possible. It allows, without slowing down the PE array:

- applications with significant working sets to take advantage of PE memory hierarchy,
- the host to be on a separate chip from the PE array and ACU, and
- the existence of multiple PE/ACU chips in a SIMD array system.

One problem not addressed here is, for multi-PE-chip systems, dealing with communication latency across chip boundaries. We will, however, briefly address this issue in the conclusion. In the following sections we describe, in turn: SIMD array architecture including memory hierarchy, VPE emulation and instruction expansion methods, implementation issues related to these methods, our own hardware implementations, experimental results comparing data locality of our approach with previous designs, and conclusions.

2 Architecture Review

Computer systems based on SIMD arrays are asymmetric, having a single host/controller and an array consisting of from a few hundred to a few hundred thousand PEs. The PEs execute synchronously code specified by the host. One consequence is that PEs do not have individual micro-sequencers or other control circuitry; rather their "CPUs" consist entirely of the datapath. Within this constraint, however, there are few limitations. The complexity of PE datapaths has ranged from the MGAP which did not have even a complete one-bit ALU [32] to the MasPar MP2 which contained a 32-bit datapath and extensive floating point support [3].

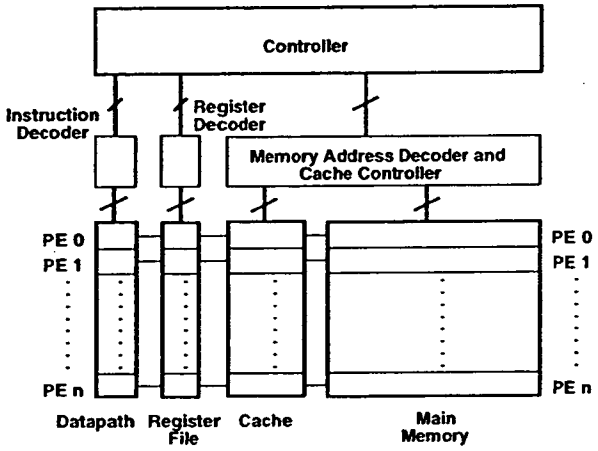


Figure 1: Shown is the PE memory design space. Note that PEs share the same controllers.

Another consequence of SIMD control is on PE memory configurations: Each array instruction generally operates on the same registers and memory locations for each PE. It is therefore possible to model the memory hierarchy, including cache, as that of a serial processor. See Figure 1.[1] For more on modeling PE memory hierarchies see [24, 1].

Current technology favors PE arrays based on memory chips. An especially promising approach has been taken by NEC with the IMAP [17]. One version is based on an SRAM chip where half of the memory has been removed and the freed-up space used for PEs. A later gener-

[1] An exception is for PE designs with a local indexing mode where registers can point to memory locations. There are usually restrictions on the indirect mode, however, so the serial model is often adequate for this case as well.

Chip	1999	2001	2003
Production DRAM	256Mb	(512Mb)	1Gb
Production SRAM	16Mb	(32Mb)	64Mb
SRAM-based IMAP-like processor	256 16 bit PEs 8Mb SRAM avail. 4KB per PE		256 32 bit PEs w/ FP 32Mb SRAM avail. 16KB per PE

Table 1: Shown is one possible development path for SIMD PE chips.

ation IMAP adds control circuitry for off-chip DRAM access [16]. The Semiconductor Industry Association Roadmap, together with a back-of the envelop calculation, yields a projection of possible per-chip capabilities of future PE chips (see Table 1). Naturally there are many design alternatives for the 2003 (and later) generation(s) with granularity being determined by application, available pins, etc. For more on these design alternatives and their evaluation see [23].

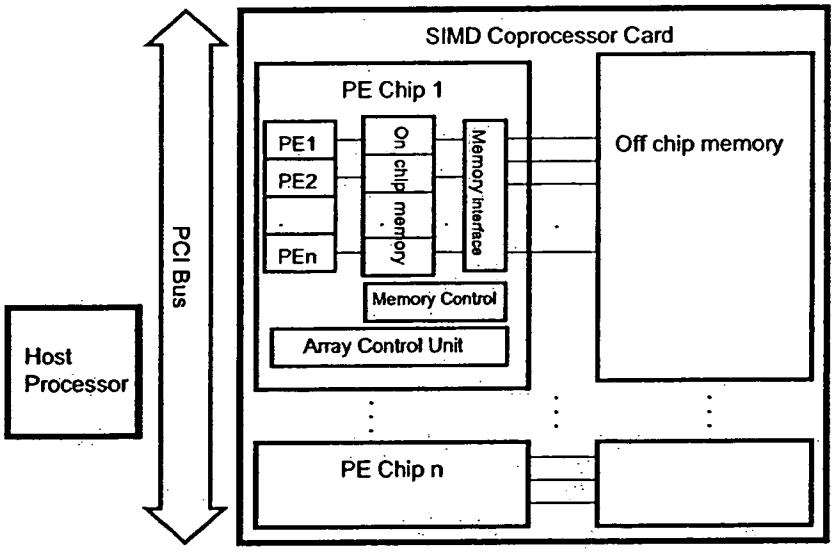


Figure 2: Shown is the implementation model with SIMD array on a coprocessor card and ACUs on the PE chips.

The system model we have in mind is shown in Figure 2. The host is a microprocessor (in our work, a standard PC) and the PE chips and memory are on a coprocessor card (e.g. on a PCI bus [12]). There are some number of PE chips on the card, each with a copy of the Array Control Unit (ACU). PEs have their own memories on chip, which can be either software or

hardware controlled cache. There is a great deal of flexibility in organizing off-chip memory; see [17] for one example. In the conclusion we briefly discuss the implications of systems where the host is on-chip as well.

3 Instruction Expansion Definitions and Orderings

In this section we detail the types of expansion, their possible execution orderings, and qualitatively evaluate those orderings. We begin by describing how VPE emulation is carried out. VPE variables are mapped *VPR* elements per physical PE, where VPR refers to the VPE/PE ratio. We refer to a single physical slice of a VPE variable across the array of physical PEs as a *tile*. In Figure 3, there are 16 tiles. The reverse mapping is also plausible; in this example the result would be each quadrant of the image being mapped to one PE. There can be large benefits for each mapping, depending on network support (see [20, 28]), but this decision has little impact with respect to either instruction issue or reference locality, the two primary topics of this article.

0,0	1,0	0,1	1,1	0,2	1,2	0,3	1,3
2,0	3,0	2,1	3,1	2,2	3,2	2,3	3,3
0,4	1,4	0,5	1,5	0,6	1,6	0,7	1,7
2,4	3,4	2,5	3,5	2,6	3,6	2,7	3,7
0,8	1,8	0,9	1,9	0,10	1,10	0,11	1,11
2,8	3,8	2,9	3,9	2,10	3,10	2,11	3,11
0,12	1,12	0,13	1,13	0,14	1,14	0,15	1,15
2,12	3,12	2,13	3,13	2,14	3,14	2,15	3,15

Figure 3: Shown is an 8x8 image with 4 PEs and 16 tiles. Each pixel labeled PE,tile.

Of more concern is how the VPE variables are laid out in PE memory. There are again two possibilities as shown in Figure 4. In (a), parallel variables are allocated contiguously, while in (b) VPR variable elements (tiles) are allocated contiguously. Which one will yield better spatial locality depends on expansion order (described next).

For each VPE instruction where there is no interaction among elements *within* a parallel

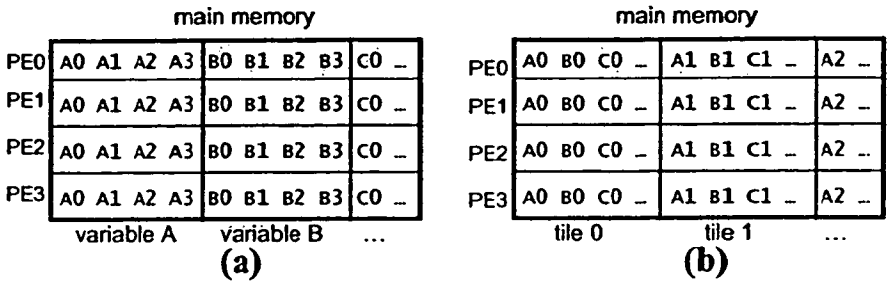


Figure 4: Shown are two possible variable layouts in memory, (a) variable first and (b) tile first, for four PEs, a VPR of 4, and some number of parallel variables.

variable–such as those for logic, arithmetic, and activity control–VPR physical instructions must be executed, one for each tile. Emulating interacting VPE instructions such as feedback and communication is more complex and can sometimes add significant overhead [21].

The best data reference locality is achieved when as many instructions as possible are executed for each tile before continuing to the next; this is because a change of tile is effectively a context switch. However, tiles *must* be swapped whenever feedback or communication instructions are encountered. This is because feedback instructions can cause control hazards in the host and because communication instructions almost always involve interaction of elements across tiles. Compilers (e.g. ICL [35, 7, 22]) can schedule instructions to minimize tile swapping. We illustrate these ideas with a code example, beginning in a high-level data parallel language:

```
// A, B, C, D, and E are 256x256 images of 32-bit
// integer pixels stored in the PE array.
// a is a 32 bit scalar stored in the host.

IntPlane(256,256) A,B,C,D,E;
int a;

B = 10;                 // 10 is a scalar known at compile time
a = 8 + 5;              // host-only operation
D = a + B;              // a is a run-time scalar
A = 7 + C;              //
if (E.ANY()) A = 1;     // ANY is a global-OR
else B = 1;             //   feedback operation
```

The following is a compiler translation of the source into tuples; `temp` is a scalar variable:

```
1. (=,B,10)
2. (+,a,8,5)
3. (+,D,a,B)
4. (+,A,7,C)
5. (ANY,temp,E)
6. (IF,temp,Tuple7,Tuple9)
7. (=,A,1)
8. (J,Tuple10)
9. (=,B,1)
10.
```

We will return to the tuple sequence later; for now we show the expansions for a single tuple: (+,A,7,C). We begin by assuming machine independence, i.e., we use the VPE array as a target and assume a VPE datapath similar to that of a standard 32-bit RISC machine. Let the 64K elements of A be assigned one per VPE and, within each VPE, to memory location 520. Let C be similarly assigned to 528. Compilation into an ordinary RISC machine assembly language yields the following VPE code. Note that these instructions belong to the array thread and so are broadcast to all VPEs. We call these VPE array assembly language instructions *VPE macroinstructions.*

```
4.1  R0 <-- 528
4.2  R1 <-- R0 + 7
4.3  520 <-- R1
```

We now begin the machine-dependent phase of compilation. Let us assume a VPR of 2 and that each VPE the PE is emulating (i.e., the tile) has been allocated 1000 bytes of memory for each physical PE. Variables A and C now occupy 2 words each in every PE's memory, one for each tile (i.e., 520/1520 and 528/1528). The following code shows one expansion; others are possible, but require more registers. We refer to this phase as *VPE expansion* and the instructions below as *PE macroinstructions.*

```
4.1.1  R0 <-- 528
4.2.1  R1 <-- R0 + 7
4.3.1  520 <-- R1
4.1.2  R0 <-- 1528
4.2.2  R1 <-- R0 + 7
4.3.2  1520 <-- R1
```

Now assume that the PEs have a 16-bit rather than 32-bit datapaths. Then the two registers must each have two accessible halves, e.g. R0 has R0_0 and R0_1, and the following code is generated. We have now finished the compilation and that these are actual the *PE instructions*. We refer to this final phase as *ALU expansion*.

```
4.1.1.1  R0_0 <-- 528              # load half-words of PE variable
4.1.1.2  R0_1 <-- 530              #   C into PE register R0
4.2.1.1  R1_0 <-- R0_0 + 7         # add low half of 32 bit immediate
4.2.1.2  R1_1 <-- R0_1 + 0 w/ carry # add high half of 32 bit immediate with carry
4.3.1.1  520 <-- R1_0              # store half-words of register
4.3.1.2  522 <-- R1_1              #   R1 into PE variable A
4.1.2.1  R0_0 <-- 1528             # load half-words of PE variable
4.1.2.2  R0_1 <-- 1530             #   C into PE register R0
4.2.2.1  R1_0 <-- R0_0 + 7         # add low half of 32 bit immediate
4.2.2.2  R1_1 <-- R0_1 + 0 w/ carry # add high half of 32 bit immediate with carry
4.3.2.1  1520 <-- R1_0             # store half-words of register
4.3.2.2  1522 <-- R1_1             #   R1 into PE variable A
```

The final code is labeled with 4 indices, corresponding to the following transformations: (1) high-level language to tuple, (2) tuple to VPE macroinstruction, (3) VPE macroinstruction to PE macroinstruction, and (4) PE macroinstruction to PE instruction. The execution order, however, follows a different nesting; in particular (2) and (3) have been reversed. Other execution orderings are also possible: this decision process is identical to that of selecting the appropriate nesting order for, say, a matrix multiplication code, and has similarly critical importance both on data reference locality and on determining the appropriate hardware support. Also, it is possible to implement "blocking" analogous to that used in compiler optimization, e.g., when accounting for cache size or multiprocessing.

We now discuss qualitatively the various possible execution orderings. Details and experimental results are presented in the next sections. There are 24 possible index orderings, each corresponding to a permutation. Most, however, make little sense and are quickly eliminated.

1. VPE macro instructions (2) can be optimized across tuples (1) but since that is a standard compiler function we do not discuss it further here. However, a complete interchange of (1) and

(2) makes little sense since macroinstruction index does not have semantic content outside the tuple. Also, this interchange generally results in many additional registers being required and for even modest VPR leads to code that will not execute. We therefore assume that (1) always precedes (2).

2. ALU expansion (4) should always come last; i.e. PE macroinstructions should not be interleaved. The following example, two versions of tuple 4, tile 2, shows why. In the first version, the three macroinstructions are executed consecutively; in the second they are interleaved. In both versions, we need to save the carry from 4.2.2.1 for use in 4.2.2.2. In the second, however, 4.2.2.1 and 4.2.2.2 are no longer contiguous and the carry must be saved across instructions. Although this is not a problem here, in general, interleaving PE macroinstructions would require saving a set of status flags for every PE macroinstruction in progress.

```
# version 1:  PE macroinstructions 1, 2, 3 are consecutive
4.1.2.1  R0_0 <-- 1528              # load half-words of PE variable
4.1.2.2  R0_1 <-- 1530              #   C into PE register R0
4.2.2.1  R1_0 <-- R0_0 + 7          # add low half of 32 bit immediate
4.2.2.2  R1_1 <-- R0_1 + 0 w/ carry # add high half of 32 bit immediate with carry
4.3.2.1  1520 <-- R1_0              # store half-words of register
4.3.2.2  1522 <-- R1_1              #   R1 into PE variable A

# version 2:  PE macroinstructions 1, 2, 3 are interleaved
4.1.2.1  R0_0 <-- 1528              # load half-words of PE variable
4.2.2.1  R1_0 <-- R0_0 + 7          # add low half of 32 bit immediate
4.3.2.1  1520 <-- R1_0              # store half-words of register
4.1.2.2  R0_1 <-- 1530              #   C into PE register R0
4.2.2.2  R1_1 <-- R0_1 + 0 w/ carry # add high half of 32 bit immediate with carry
4.3.2.2  1522 <-- R1_1              #   R1 into PE variable A
```

This leave three possible permutations: 1234, 1324, and 3124.

3. That 1234 is also not viable is shown below. Obviously this code is incorrect and different registers would be used by the 4.x.2.x instructions. The problem is that a VPR sets would be required.

```
4.1.1.1  R0_0 <-- 528
4.1.1.2  R0_1 <-- 530
4.1.2.1  R0_0 <-- 1528
```

```
4.1.2.2  R0_1 <-- 1530
4.2.1.1  R1_0 <-- R0_0 + 7
4.2.1.2  R1_1 <-- R0_1 + 0 w/ carry
4.2.2.1  R1_0 <-- R0_0 + 7
4.2.2.2  R1_1 <-- R0_1 + 0 w/ carry
4.3.1.1  520 <-- R1_0
4.3.1.2  522 <-- R1_1
4.3.2.1  1520 <-- R1_0
4.3.2.2  1522 <-- R1_1
```

The two final permutations are both interesting and are shown for two tuples:

4. In 1324, the priority is on executing VPE macroinstructions consecutively. The resulting macrocode for two tuples is shown.

```
# VPE Expansion First
3.1.1   R0  <-- 524      # get B from PE memory
3.2.1   R0  <-- R0 + a   # add scalar a to PE reg
3.3.1   532 <-- R0       # write back D
3.1.2   R0  <-- 1524     #
3.2.2   R0  <-- R0 + a   #   Tile 2
3.3.2   1532 <-- R0      #
4.1.1   R0  <-- 528      # get C from PE memory
4.2.1   R0  <-- R0 + 7   # add scalar 7 to C
4.3.1   520 <-- R0       # write back A
4.1.2   R0  <-- 1528     #
4.2.2   R0  <-- R0 + 7   #   Tile 2
4.3.2   1520 <-- R0      #
```

5. In 3124 the priority is on executing as many macroinstructions for each tile as possible before executing the same macroinstructions for the next tile. This requires that VPE macroinstructions be interleaved. This is generally possible; the exception is when there are dependencies *within* the VPE macroinstruction, which happens exactly when there is interPE communication or when feedback is collected (usually for a flow control decision in the host). The resulting code for two tuples is shown.

```
# Execute within VPEs (tiles) for as long as possible
3.1.1   R0  <-- 524      # get B from PE memory
3.2.1   R0  <-- R0 + a   # add scalar a to PE reg
3.3.1   532 <-- R0       # write back D
4.1.1   R0  <-- 528      # get C from PE memory
```

```
4.2.1   R0  <-- R0 + 7   # add scalar 7 to C
4.3.1   520 <-- R0       # write back A
3.1.2   R0  <-- 1524     #
3.2.2   R0  <-- R0 + a   #
3.3.2   1532 <-- R0      #    Tile 2
4.1.2   R0  <-- 1528     #
4.2.2   R0  <-- R0 + 7   #
4.3.2   1520 <-- R0      #
```

4 Implementation Issues

We are left with two viable expansion orderings: VPE first and Tile first. We discuss these now with respect to (i) hardware support for instruction issue and expansion and (ii) reduction hazards. The latter are defined as control hazards where program flow is dependent on feedback from the array, usually through a reduction operation such as ANY or COUNT.

VPE Expansion First

The host sends the VPE macroinstructions to the array as they occur in the main thread. The ACU does both expansions. See Figure 5a. Instructions are generated from each VPE macroinstruction through incremental changes to the instruction fields as shown in the previous section. The advantages of "VPE first" are large expansion factor and simplicity: the ACU can easily generate the PE instructions with no loss in cycle time and with only a few extra pipeline stages. Since the array executes all instructions as they occur in the host code, handling reduction hazards requires only waiting for array-host-array turn-around.

In-Tile First

The host handles the tiling expansion and sends only the PE macroinstructions to the array. The ACU does only the ALU expansion, (see Figure 5b). Instruction issue is a problem. Historically, since ALU expansions were typically 16 to 32, this was just adequate to keep the array busy. Later, large issue buffers were used to deal with issue variance. These large buffers needed to be flushed on reduction hazards. Now-a-days, however, an ALU expansion of between 1 and 4 is more likely; not nearly enough to hide the host's instruction issue overhead.

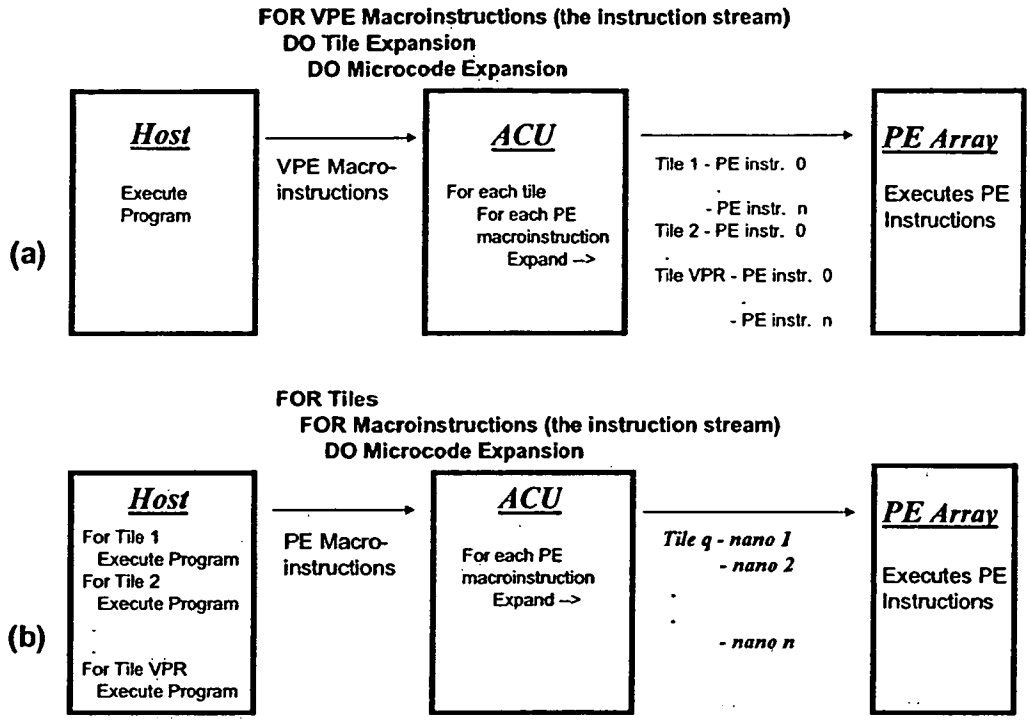


Figure 5: (a) Both expansions are done by ACU; (b) Only ALU expansion is done by ACU.

Solution

The problem is therefore that neither ordering provides both viable instruction issue and reasonable data locality. Our solution is to use the "within tile first" expansion but move much of the control to the ACU. We begin with some definitions. We use the compiler term *basic block* to refer to the sequences of PE instructions that can be executed within a tile before execution on a new tile must be initiated. We use the term *ACU instruction* to refer to directives by the host to the ACU, but not to the PEs.

From our original example, the first five tuples form a basic block. It is shown together with the corresponding VPE macroinstructions (keeping our previous memory allocation):

```
  TUPLES          MACROINSTRUCTIONS
1. (=,B,10)       R0  <-- #10
                  524 <-- R0
2. (+,a,8,5)
3. (+,D,a,B)      R1  <-- R0 + scalar(a)
                  532 <-- R1
4. (+,A,7,C)      R2  <-- 528
```

```
                    R3  <-- R2 + 7
                    520 <-- R3
5. (ANY,temp,E)     ANY(536)
                    DONE
```

There are several things to note. One is that Tuple 3 has no corresponding macro instruction since it is executed entirely within the host. A second is that, although the scalars 10 and 7 are known before run time, the value of scalar a is not. Finally, the ACU needs to know that the basic block is finished so it can loop back or look for more work. See Figure 6 for a sketch.

The host begins program execution by downloading a fraction of the program's basic blocks into the ACU's macroinstruction memory. The host then executes its program. However, rather than broadcasting each PE instruction, or even each macroinstruction, as needed, it only needs to broadcast the appropriate directives. These include:

- Configuration information, such as VPE memory allocation
- The address of the next basic block
- The run-time-computed scalars and addresses for insertion into the instruction stream
- Actions to be taken associated with the basic block, such as "execute 10 times," "look for next basic block," or "examine feedback result and execute one of the two following basic blocks depending on result."

The host thus executes the following code:

```
1. ACU(CONF,2,1000);       // tell ACU there are 2 tiles of 1000 bytes
2. ACU(BB,134,1);          // tell ACU to execute BB at 134 once
3. a = 8 + 5;              // compute run time scalar
4. ACU(IMM,a);             // compute run time scalar
5. temp = ACU(FEEDBACK);   // get feedback value from ACU to temp
// do test and tell ACU which BB to do next
6. if (temp == TRUE) ACU(BB,293,1);
7. else ACU(BB,572,1);
```

ACU() is the generic function used by the host to talk with the ACU. For efficiency, ACU() is in-lined and done with memory-mapped user-mode I/O [12]. While the host is executing the code just shown, the ACU is initially idle and waiting for instructions to come through its INFIFO. The first ACU directive, `ACU(CONF,2,1000)` , initializes the ACU's internal registers.

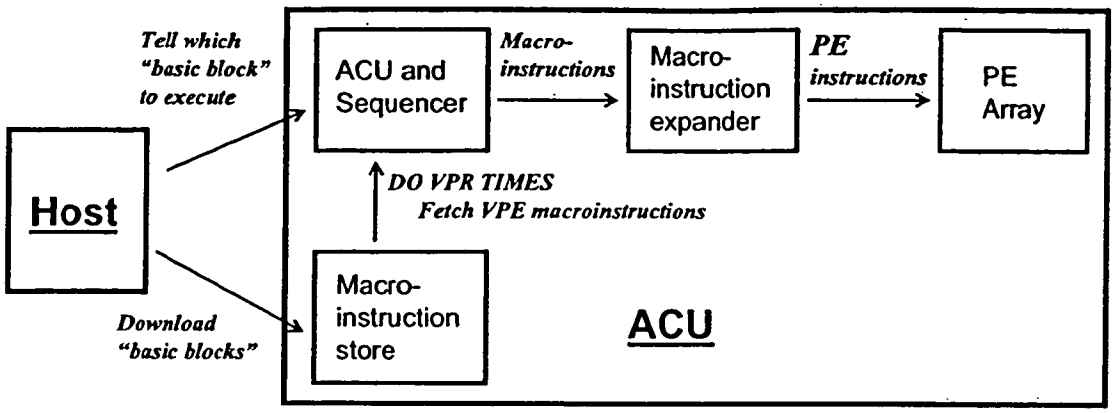


Figure 6: Basic block expansion by the ACU.

The second begins macroinstruction fetch, expansion, and PE instruction issue. The ACU recognizes when a macroinstruction needs a run-time scalar or a dynamically computed address and reads it from its INFIFO. The ANY macroinstruction requires that the ACU keep track of the current feedback value; when the entire basic block has been executed VPR times, the feedback value is put into its OUTFIFO. Finally, the ACU looks to the INFIFO for the next basic block. The ACU can also deal with reduction hazards directly; this is described below.

5 ACU Implementation

5.1 Functions and Components

A schematic of the hardware design is shown in Figure 7. We are exploring several different implementations (see Section 5.2), but they all have the same basic characteristics. At the highest level, the system consists of a host PC which communicates via a PCI interface with the board containing the array. On the array board are some number of ACU/PE chips, additional DRAM for PE data, and a video interface. PEs are as described in [23]: they have some amount of on-chip SRAM and an interface to off-chip DRAM. The PE granularity and memory sizes are technology and application dependent.

The keys to implementing the ACU are that the operating frequency be comparable to that of the PEs and that a PE instruction be issued every cycle. Minimizing the length of the

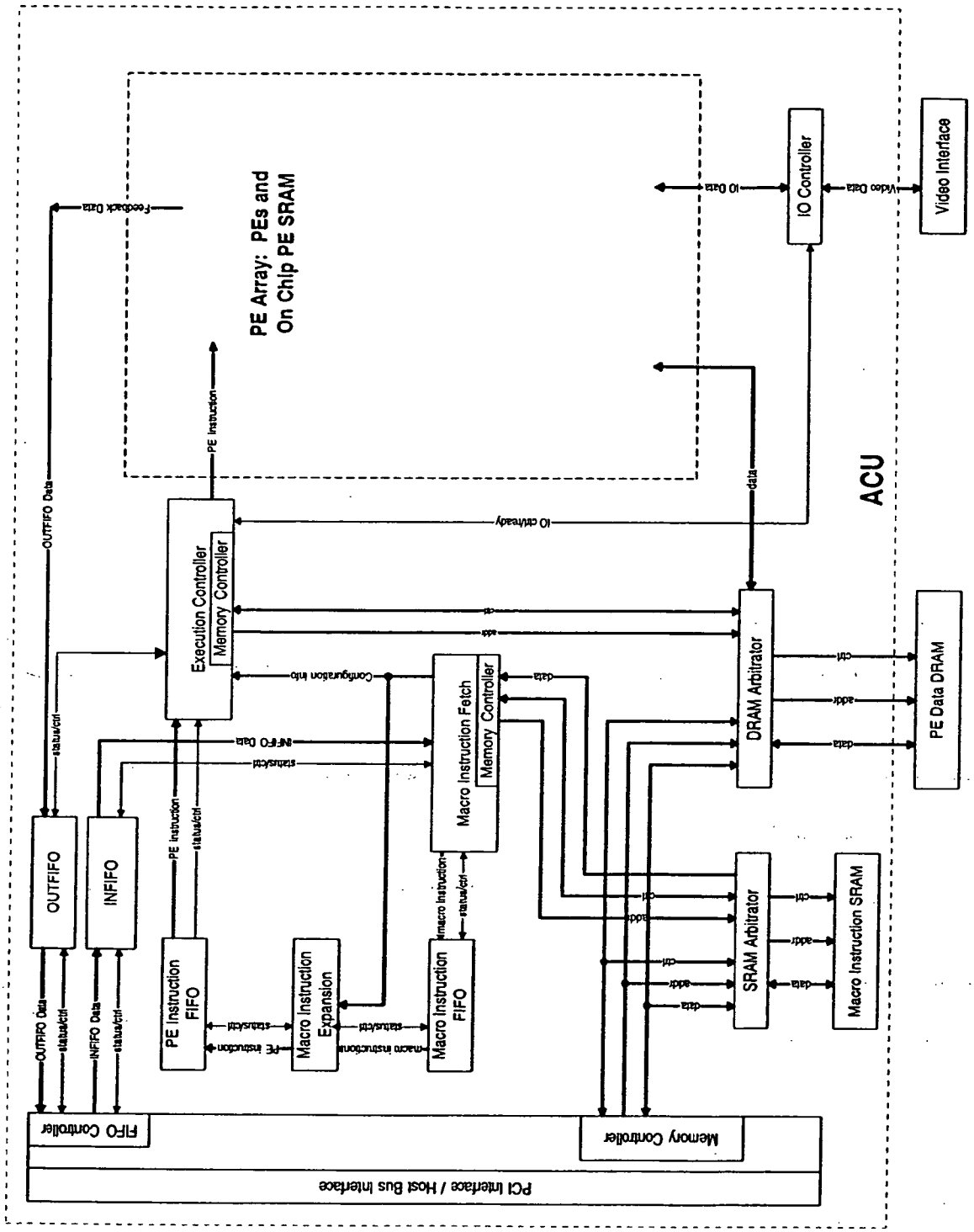


Figure 7: Shown is the block diagram for the ACU together with interfaces to host, PE array, and I/O.

instruction issue pipeline is also a goal, but much less important than the first two. We now enumerate some functions required for ACU operation and the hardware units that carry them out.

1. *Host sends basic blocks of macroinstructions to ACU.* Done by PCI interface and SRAM arbiter.
2. *Host sends and receives PE data.* Done by PCI interface and DRAM arbiter.
3. *Host sends ACU instructions and dynamic scalars.* Done by PCI interface and INFIFO.
4. *Host receives feedback data.* Done by OUTFIFO and PCI interface.
5. *Sequence PE macro instructions.* Done by macroinstruction fetch unit.
6. *Expand macro instructions into PE instructions.* Done in two stages: the macroinstruction fetch unit does tile expansion through iterations of basic blocks. The macro to PE code expansion is done by the macroinstruction expander.
7. *ACU instruction execution.* The macroinstruction fetch unit is the "controller controller."
8. *Dynamic scalar integration with PE instructions.* Done by the macroinstruction fetch unit.
9. *PE instruction issue.* Done by the PE instruction FIFO and the execution controller.

The core of the ACU is the Macroinstruction Fetch unit. It executes two ACU instructions: *configure* and *execute basic block.* Configure tells how many times to execute each basic block and the size of the PE memory per tile. Execute gives the address of the next basic block in the Macroinstruction SRAM. The Macroinstruction Fetch unit also handles dynamic scalars by recognizing which macroinstructions need them, fetching the values from the INFIFO, and integrating the values into the macro instructions.

The hardware details are as follows. There is a program counter for macro instruction sequencing and a tile counter for determining when execution of the basic block is complete. We assume that the size of data allocated to each PE tile is a power of two. This allows address calculation of the PE data to proceed without addition as the tile number can be merged with the macro instruction data address to generate the correct data address for each tile. The Macroinstruction Fetch unit has a three stage pipeline, including writing to the

Macroinstruction FIFO.

The Macroinstruction Expansion Unit also has a three stage pipeline including writing to the PE Instruction FIFO. Its functions include sequencing the expansion and merging the register file addresses.

Two more stages are needed by the execution controller and PE decoders yielding a maximum of eight stages between macro nstruction fetch and PE instruction issue to the PE array. Depending on instruction distribution timing characteristics of the PE array, additional stages may be required there as well. Finally, the PEs themselves have three stage pipelines.

5.2 Implementation Status

We have a large number of working PE designs which can be interchanged depending on the granularity required by the application [23]. There are also a number of computer vision applications written in the data parallel language ICL [22]. Host interface software has been developed [12]. The ACU has been implemented in Verilog and simulated. Not yet finalized is the macroinstruction expansion unit which depends on the selections of the PE design and inter-PE routing mechanism.

We are currently investigating three implementations based, respectively, on (i) a Chip Express .35 micron gate array, (ii) a Nallatech PCI board and plug-in module which together contain a PCI interface, 3 Xilinx FPGAs (2 Virtex 1000s and 1 Virtex 300, both with -4 speed grade), 4MB SRAM, and 4MB SDRAM, and (iii) a .25 micron standard cell process from LSI-Logic. The Nallatech-based implementation will be a working system while the Chip Express and LSILogic implementations are currently only intended as virtual prototypes. The Chip Express and LSILogic implementations have been synthesized; the Xilinx/Nallatech implementation has been synthesized and gone through initial place-and-route. We will soon also be looking at a virtual prototype implementation based on the LSILogic G12 .18 micron standard cell process.

Some implementation details follow; preliminary timing and area results are in Table 2. We assume 8 bit PEs with multiplier and 32 bytes of register file. In the Nallatech/Xilinx system, we use the Virtex 300 and the SRAM for the ACU and the 2 Virtex 1000s and the DRAM for the PE array. Better performance could easily be achieved with higher speed-grade or newer FPGAs (e.g. Virtex E-series). For the Chip Express system, we assume the CX3551 master slice. Achieving 500MHz is probably unrealistic, but we believe that 250MHz is not, even with more complex PEs. A system based on the LSILogic or similar standard cell process should be able to approach the higher operating frequencies, however, and also allow optimization of the PE memory hierarchy.

	Nallatech/Xilinx 2 Virtex1K -4 FPGAs	Chip Express .35u Gate Array	LSILogic G11 .25u Standard Cell
ACU Timing	19.2ns	2.14ns	.79ns
PE Timing	21.5ns	1.95ns	.85ns
# of PEs	128 in system	64 per chip	256 per chip
per PE SRAM	128 bytes	512 bytes	50% of chip
per PE DRAM	16KB	TBD	TBD

Table 2: Preliminary characteristics of three implementations. Per PE SRAM is on chip; per PE DRAM is off chip.

6 Impact on Memory Performance

6.1 SIMD Program Characteristics and their Implications

A more realistic view of SIMD array code

For memory performance, the two critical application characteristics are the image size, from which the VPR immediately follows, and the distribution of reduction hazards and communications from which the basic block size follows.

Reduction hazards tend to be rare: where reduction hazards do occur, they are generally termination conditions of relaxation-based algorithms. Although this is an important paradigm, even here, reduction hazards rarely occur more often than once per 10's of thousands of PE

instructions.

Communication, on the other hand, tends to be either frequent (occurring every few VPE macroinstructions) or rare (occurring less than every several hundred VPE macroinstructions). An application with the former characteristic is correlation-based stereo matching. An example of the latter is a focus-of-expansion computation. Some applications, e.g. a region-merging segmentation algorithm, run in phases between the two.

Since communication instructions generally include transfers *among* all tiles in a parallel variable, all preceding VPE instructions must be completed before they begin. For the same reason, communications must complete before the following instruction begins. We can therefore view each communication instruction as its own basic block.

Implications on working set size

For large basic blocks, the working set size per physical PE approaches that of a single VPE. For small basic blocks, the working set size includes the working sets of all VPEs being emulated by the PE. VPE working sets being equal, instruction sequences with small basic blocks can result in working sets up to VPR times greater than sequences with large basic blocks. Also, the bigger the VPR, the bigger the basic blocks needed for the effect of the VPR on the working set size to diminish.

Where this has the potential to be a serious problem is in applications with frequent communication, especially if the images are large (resulting in large VPR). Fortunately, it appears to almost universally the case that applications with frequent communications operate on very few images at a time. The paradigmatic high-communication tasks are convolutions and correlations for matching: both operate on only a few images at a time. The design of memory hierarchies for high-communication tasks reduces to satisfying a simple condition: provide enough on-chip memory to hold the images being correlated. Fortunately, current technology makes satisfying this condition straightforward for all but the very largest images. In this case, however, the general solution is to process parts of the image separately.

Implications for instruction issue

The worst case for instruction issue is where the basic block size is one. Here, however, the "basic-block" method simply reduces to "VPE expansion first." As described above, VPE expansion first does fine with instruction issue (its problem being reference locality).

Dealing with reductions in the ACU

As with most programs, SIMD array codes are built on loops. A substantial difference with corresponding high-level language codes is that each VPE macroinstruction generally replaces two inner loops (rows and columns). The remaining loops in the dataparallel code therefore have the characteristics of "outer" loops in sequential computations: they are executed relatively few times.

It is usual that basic blocks get used several times in close succession. However, it is rare that a basic block gets used iteratively: most often two or more basic blocks are executed in a loop with communication being interspersed. As a consequence, the ACU support for iterative execution of basic blocks is rarely used. It would certainly not be very difficult to augment the ACU to execute a basic block stream. However, there does not seem to be a reason to do so at this time by the same reasoning as immediately above.

6.2 Results and Discussion

Hardware Model

- Number of boards – We assume a single board SIMD array; more are possible, but exacerbate communication difficulties.
- On-chip memory – We assume the upper bound of on-chip memory size to be half that of an SRAM chip (with the other half of the chip allocated to PE logic). In current technology this comes to 16Mb per PE chip, growing to four times that in the next two to three years. The low-end (e.g. in an FPGA-based system) could be substantially smaller.
- Off-chip memory – With current memory densities and our applications, having sufficient off-chip memory to hold all application data should not be an issue.

- Number of PE chips – Between 1 and 8 are reasonable for a standard sized PCI board.
- Number of PEs per chip – From our own results and those from NEC, 256 8-bit PEs per chip are viable using three-year-old technology. In the next two to three years, putting 512 PEs with 32-bit datapaths and floating point support on a chip should be possible.
- Size of cache per PE – From the previous, possible cache sizes per PE range from 128 bytes with a current FPGA-based design to 16KB for a projected standard-cell design.
- Total cache size – This is an important parameter for communication intensive applications. The range for standard cell designs is 2MB for a current single chip design to 64MB for a future eight chip system. This is enough to hold from 8 to 256 medium resolution images.
- Cache parameters – A previous study has shown the benefits of high associativity and small block sizes for PE caches [24]. We therefore assume an associativity of eight and a block size of four.
- Register File Size – Because many of the parameters normally stored in registers have to do with control which a SIMD PE does not need to deal with, SIMD PEs can have smaller register files. We have found that having more than 16 32-bit registers does not help. This is therefore what we assume here.

Software model

We examine two expansion modes, VPE first and basic blocks. We use the appropriate memory layout for each, i.e. variables together for VPE first and VPEs (tiles) together for basic blocks.

Applications

1. BMAll – The low- and intermediate-level parts of ARPA IU Benchmark II [36]. This is an integrated series of tasks including convolutions, segmentation, convex hull, and many others. Substantial computation interspersed with communication. Both integer and floating point computations
2. motion256 – Depth from correspondence [14]. Contains substantial floating point computations with very little communication.

3. spiral3 – Correlation-based correspondence matcher. Dominated by communication. Integer computation only.

4. prewitt – A region-based edge-grouping line finder, based on the Burns algorithm [6]. Uses both a substantial number of integer operations and communications.

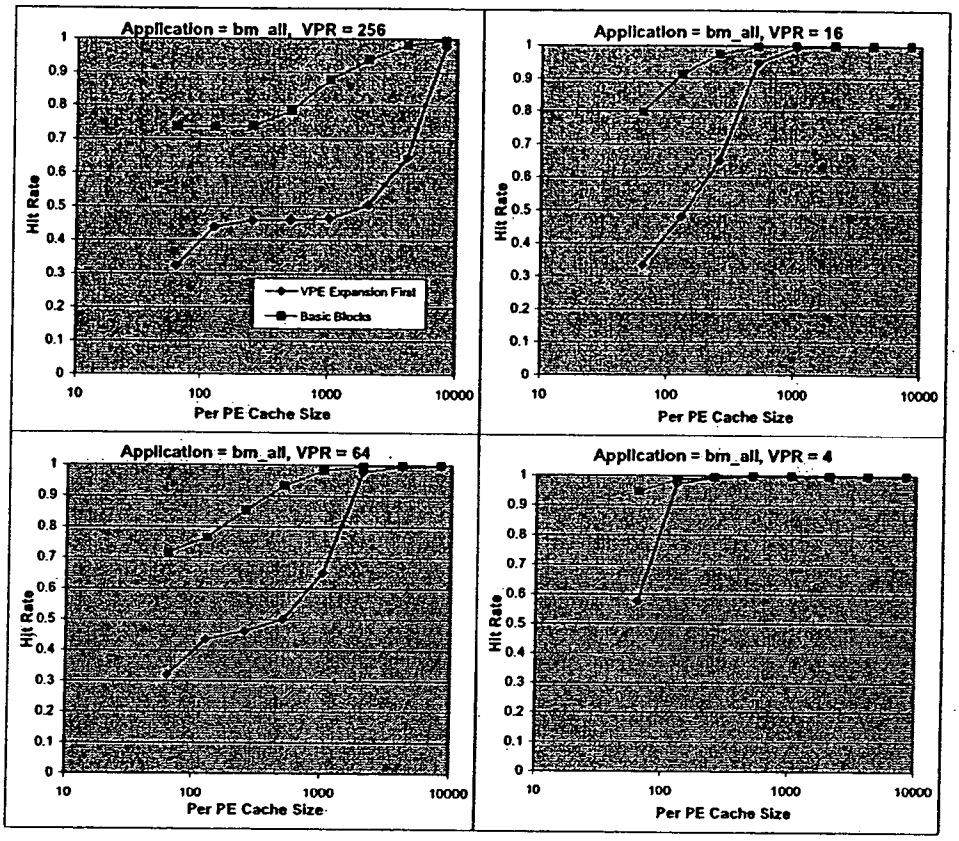


Figure 8: For BMAll, hit rate versus cache size while varying expansion method and VPR.

Results

We compare hit rates of VPE expansion first (VEF) with basic blocks (BB). We vary application, VPR, and cache size. In the first series of graphs (Figure 8), we show BMAll for four different VPRs. In the second series (Figure 9), we show all four applications for a VPR of 64. In both cases, the per PE cache size ranges from 64 bytes to 8KB. The other hardware and software parameters are as described above. These results are representative of the several hundred thousand datapoints we have generated and whose detailed analysis will be the subject of a future study.

VPR captures the relationship between image size and array size: For example, a VPR of 64 represents an image size of 64K pixels and an array size of 1K PEs as well as an image size of 256K pixels and an array size of 4K PEs. Recall that the VPR also represents the maximum difference in working set size between VEF and BB.

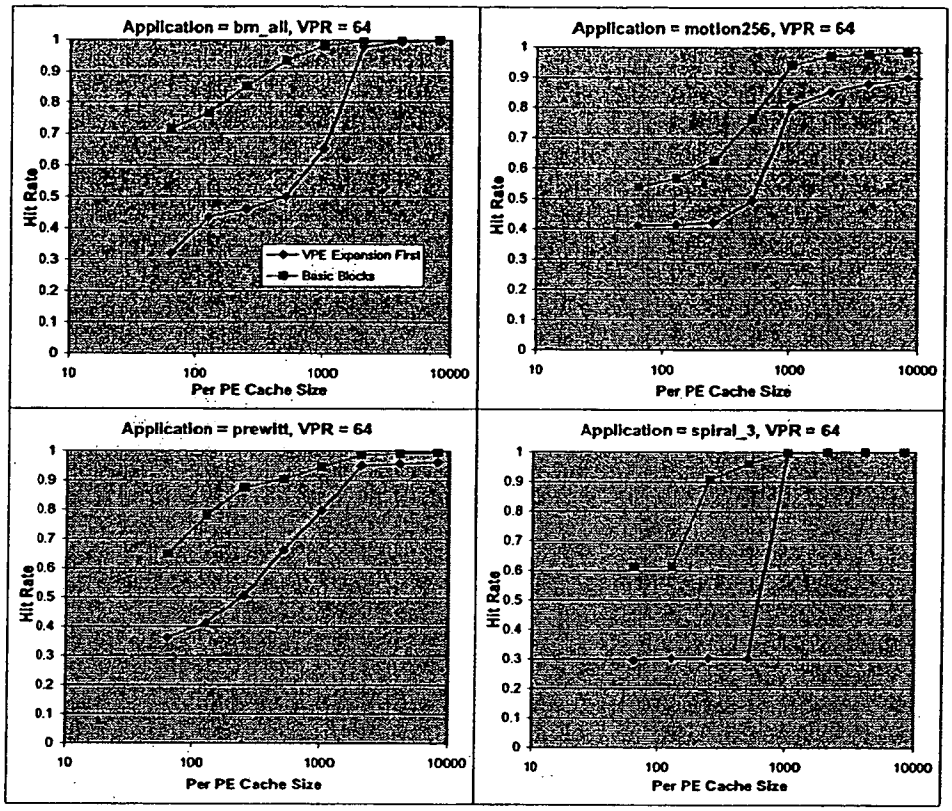


Figure 9: For VPR = 64, hit rate versus cache size while varying expansion method and application.

The superiority of the basic blocks approach is clearly shown. In BMAll, cache must be roughly 4 times larger for VEF than BB to get good performance, i.e. a hit rate of over 90%. Also, BB approaches that performance more rapidly than VEF: BB achieves a 70% to 80% hit rate with very small caches even for large VPR, while VEF performance declines precipitously with a slight reduction in cache size.

In the second figure, we see the effects of application differences. BMAll and Prewitt are both comparatively long codes with a number of different phases and a large variation in

basic block sizes. Therefore, Prewitt and BMall both see a gradual increase in working set for both expansion methods, but with a more rapid increase for BB. Spiral3, although similarly computationally intensive to the other applications, is a very simple code with very small basic blocks. This explains the sharply defined working set and the simple tracking of performance of VEF and BB. We also see that this is an example of an effect described earlier: that applications with very frequent communication also tend to have small working sets. Motion256 on the other hand, has almost no communication, and therefore very large basic blocks, as well as the largest working set of the applications studied here. A 1K per PE cache is sufficient to achieve a 93% hit rate for BB, but even an 8KB cache is not enough to reach 90% for VEF.

7 Conclusion

We have described the issues involved in instruction selection and issue for SIMD processor chips. Our solution involves preloading basic blocks of VPE macroinstructions and expanding them in a two stage process: by tile and by microcode. In this way we are able to keep up with high speed PEs and also maintain the locality in the PE array data stream.

Our solution does not lose effectiveness when applied to multiple ACU/PE chips: the ACU is simply replicated along with each PE chip. As interPE communication is more likely to become a bottleneck in these designs, many recent arrays have been one dimensional. Other solutions involve asynchronous transfers and other latency hiding techniques well known in multiprocessors.

References

[1] Allen, J. D., and Schimmel, D. E. Issues in the design of high performance SIMD architectures. *IEEE Trans. on Parallel and Distributed Systems 7*, 8 (1996), 818–829.

[2] Bishop, B., Zhang, Y., Acken, K., Irwin, M., and Owens, R. Three dimensional graphics algorithms on the Micro-Grain Array Processor-II. In *Proc. of Computer Architectures for Machine Perception* (1997), pp. 204–208.